# Design of a Real-Time Embedded Control System for Quantum Computing Experiments

by

Richard Rademacher

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Physics (Quantum Information)

Waterloo, Ontario, Canada, 2020

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

This thesis describes the design of a real-time control system for trapped ion quantum computer experiments. It is framed in the context of the QuantumIon project, a project at the University of Waterloo's Institute for Quantum Computing that aims to provide a scalable, remote-operation ion trap for a wide variety of quantum research without the need for 'expert' ion-trap knowledge. The target users span the range of ion-trap researchers, algorithms researchers, performance benchmarking researchers, and quantum simulation researchers.

The control system features a user programming language, remote access to a compiling server, a sub-nanosecond time sequencing engine, arbitrary waveform generation for pulse shaping, and fully adjustable internal parameters. This platform affords the user extraordinary flexibility for many research use cases without requiring physical access. High-speed precision timing is achieved through the use of FPGA technology, while internal consistency (necessary for usability by non-experts) is achieved through an abstraction layer approach. Supercomputing-grade network infrastructure is employed to meet the strict timing requirements. An extensive suite of calibration tools and results is available to monitor machine-dependent parameters of the experiment. A sophisticated symbolic algebra system is used to create powerful calculations of precision timing sequences. Extensive automation is employed to remove the need for physical access, thus providing quantum computing to a wide audience. Under this model even the lowest-level control is available to support innovative new designs, while a "library" of pre-defined sequences is also available to leverage "best practice" gates for those wishing rapid results. Finally, the user language itself is designed to be portable, allowing bindings to current popular classical languages such as Matlab and Python, and can be expanded for use in quantum-specific languages such as Cirq [1], Quill [2], and QASM [3].

Through this approach the control system for QuantumIon is a flexible, powerful, scalable, and robust platform that is expected to be in use for a long time.

**Acknowledgements**

I would like to thank all the people who made this thesis possible.

Special thanks to the members of the IQC Ion Trap Groups: In random permutation: Noah Greenberg, Brendan White, Pei Jiang Low, Roland Hablùtzel, Matt Day, Gilbert Shih, Sainath Motlakunta, Nikhil Kotibahskar, Manas Saijian, and Fereshteh Rajabi. Also thanks to the undergraduate co-op students at our group who I've worked closely with: Asmae Benhemou, Kieanna Fana, Jason Elsted, and Imad Syed. Thanks of course to Matt Cooper for Information Technology support, Virginia Frey and James Dooley for comments on this manuscript.

None of this is possible for any of us without my advisors Prof Rajibul Islam, Prof Michal Bajcsy, Prof Josef Emerson, Prof Raymond LaFlamme, and especially Prof Crystal Senko, whose unbelievable patience with my 'process' made everything possible.

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Canada First Research Excellence Fund (CFREF).

## Dedication

This is dedicated to the one I love most...you know who you are.

# Table of Contents

# List of Figures

# List of Tables

# List of Listings

# Abbreviations

This document is incomplete. The external file associated with the glossary 'abbreviations' (which should be called `output.gls-abr`) hasn't been created.

Check the contents of the file `output.glo-abr`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `output.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "output"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "output"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

# Chapter 1

# Introduction

Ion traps are one of the most promising techniques for realizing a physical apparatus for quantum computing. With their long decoherence times, high scalability, and mature experimental techniques, ion traps can provide the platform for long-term research. As an experimental platform, there is much that goes between the circuit-level descriptions of quantum algorithms ([4], Ch 4) and the physical realization of a quantum computing machine.

## 1.1 Quantum Computation

The idea of using the laws of quantum mechanics as a platform for computation has been around since the 1980s[5]. Later, the use of a quantum process as a means to accelerate particularly difficult classical problems was brought to light in the form of uniquely-quantum algorithms by Shor[6], and by Grover [7]. These two algorithms stimulated an explosion in research to the applications of quantum computation for fields like error correcting codes, optimization, cryptography, and many more problems.

In contrast to classical computing, in the field of quantum computing the software has moved much more rapidly than the construction of actual machines to carry out these algorithms. The present state-of-the-art in the design of the physical machine is being performed around the world by experimental physicists, and engineers.

Quantum computing takes advantage of two unique properties of quantum mechanics, the field of physics that describes the world at the atomic scale. The first of these, superposition, allows quantum information to be treated as having many values (states)

simultaneously. It is only when the quantum information, the *qubit*, is measured that a single value is returned. However, prior to measurement, qubits may interact with each other as if they had several values simultaneously. This property can provide a means to parallelize mathematical operations.

The second property, entanglement, is another uniquely quantum property that has no analog in classical computation. In this case two qubits are manipulated such that they follow each other: manipulations to one qubit affect the other. Entangling can form an analog to *copying* of information (something not allowed in quantum information).

These two properties, superposition and entanglement, are the primary workhorses of quantum computation. Through manipulation of quantum bits, these two properties, along with the preparation of initial qubit state and the readout of results, can be exploited to perform a wide range of otherwise difficult computing problems.

## 1.2 The QuantumIon Project

QuantumIon is a project at the Institute for Quantum Computing (IQC) at the University of Waterloo. The goal of this project is to provide the infrastructure, equipment, software, and staff for a platform that enables a wide range of researchers to perform quantum experiments. QuantumIon is a real, physical quantum computer; not a simulator[1].

The intended user of QuantumIon is the university research group, generally at the graduate or professional level. This decision forced the group to consider more advanced algorithms, protocols, and performance than had been seen in other quantum computers connected to the Internet. QuantumIon is indeed intended to be connected to the Internet, and it is assumed that the users wish a level of privacy for their work, therefore security is considered early on. QuantumIon also takes great pains to be a stable platform for these researchers; a platform that is constantly under modification would fail to be useful.

The QuantumIon team consists of two co-principle investigators: Prof. Crystal Senko and Prof. Rajibul Islam (both of IQC). Additionally, the team consists of a postdoctoral fellow and two Masters students leading the optics design and mechanical & vacuum designs. These five comprise the current core QuantumIon team, and engage in intense collaboration during design. QuantumIon also heavily employs the Waterloo cooperative education program for undergraduates. Approximately six different co-op students have

[1]Note that *quantum simulations* are distinct from simulations of quantum systems on a classical computer. Quantum simulations are still experiments on a physical platform, and are possible on QuantumIon.

contributed to support, analysis and design roles. In particular two co-op students were involved in coding some of the FPGA functions described in Chapter 3.

## 1.3 Author's Contributions

This research took place during the design and planning phase of QuantumIon. That is to say, this thesis reflects the plans for a control system, but not the results of implementing that design; that work is in-progress as of this writing. During the design phase, a great deal of emphasis was placed on ensuring a good understanding of the different types of researchers, their needs, and a representative sample of different quantum algorithms, experiments, and protocols.

The author's work on this project began around September of 2018, after discussions with Prof. Crystal Senko about ideas for an online quantum computer. The author had significant experience with electronics, embedded electronic control systems, communications, supercomputer design, and Field-Programmable Gate Array (FPGA) systems. These experiences were leveraged in QuantumIon.

This thesis focuses on the control system, electronics, programming and computer design, of which the author was the leader and primary designer. An approximate breakdown is shown in Figure 1.1.

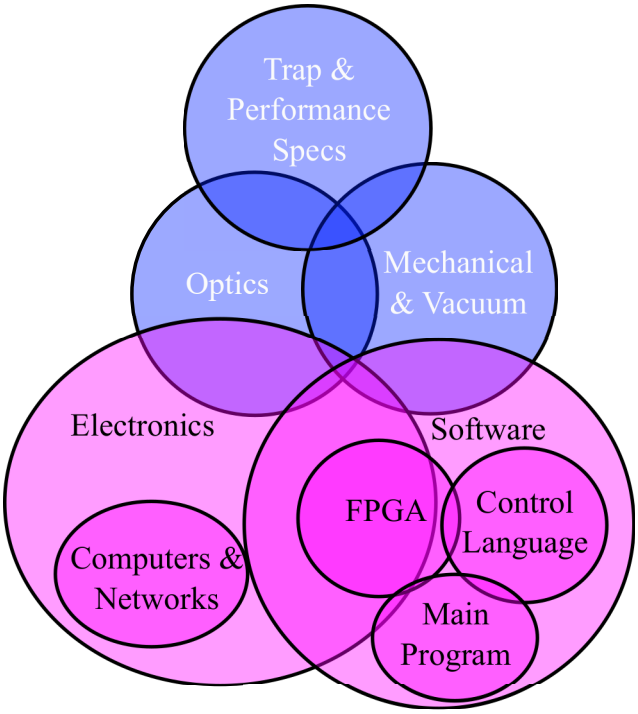


Figure 1.1: Approximate Venn diagram of author contributions. Electronics design and computer design, and the main program design were the sole effort of the author. The control language, and FPGA execution engine design were collaborations between the author and Prof. Senko. The branching logic was a collaboration with the author, Prof. Senko, Prof. Joseph Emerson, and Prof. Joel Walman of IQC. Items in blue are by other QuantumIon teams.

## 1.4 Motivation

Before detailing the architecture and implementation of the QuantumIon system, it is useful to discuss the project's motivation.

### 1.4.1 A Shared Quantum Resource

QuantumIon is designed firstly to be a shared resource for experimental quantum information. The requirement for sharing has lead the research in this thesis to alter many

traditional paradigms of experimental physics labs. These will be shown to be deliberate alterations, and with good reasons.

Current trends in trapped-ion quantum research follow a process shared by most physical laboratories, such as those for atomic and molecular optics, high-energy physics, and materials science. The equipment in these laboratories is extremely expensive, and requires researchers trained not only in experimental physics (which can be quite different from theoretical physics), but also with the intricacies of the specific apparatus. Likewise, the apparatus itself is generally designed for frequent, rapid modification in order to try new techniques and overcome obstacles. The cost of such labs is high, because the techniques require pristine control of atomic behavior (and thus expensive components), manufacture of the apparatus, and calibration.

The experimental community for ion trap quantum computers has enjoyed great success through the results at groups like National Institute of Standards and Technology (NIST)[8][9], Joint Quantum Institute (JQI)[10][11][12], Sandia National Laboratories[13], and the University of Innsbruck[14][15][16], among many others. These successes have gained the interest of theorists in many areas of quantum information, who benefit from access to an experimental platform. Typically theories are tested using a collaboration between a theory group and an experiment group. This close pairing has many advantages, but presently there are more theories to be tested than ion trap groups to test them.

The goal of QuantumIon is to bring the power of a trapped ion quantum computer directly to the hands of the current body of researchers. The target user is the university research group, and commercial efforts, but the expectation is that these users will already be proficient in quantum information science. In the pages that follow this thesis will show a real, implementable hardware and software platform to make this goal a reality.

### 1.4.2 Automation

As part of its success, the ion trap apparatus has matured to the point where automation is possible. If such automation is achieved, the theorist can *directly program* the quantum computer, obtain results, and post-process and interpret the data. In this sense, physical access to the quantum computer will become unnecessary. Automation, to the point of remote use, that is also useful to cutting-edge research is not easily achieved. The system must be flexible, stable, and repeatable.

Flexibility means that considerable care must be taken to foresee as many different uses of the machine, and then ensure that the user has considerable control over the machine

parameters that make each use case possible. Examples of such controls are the intensity of lasers, sideband content, magnetic field strength, and so on.

The system must be stable to the outside user; stability means not only control of external environmental factors, but the machine itself must be in a static, consistent condition. This imposes an important restriction on how often the machine might be upgraded. This is also called a High-Availability system: one that is mostly ready to work. The restriction on frequent modifications is outweighed by the benefit of consistent, long-term use.

Repeatability means that the same experiment run hours, or days apart should give the same results[2]. However, all physical apparatus suffer from changes to parameters as it interacts with the outside world. To this end, an extensive calibration scheme is required, and so is a sophisticated means of using this calibration to decouple the machine from aging, short-time fluctuations, and the environment. Isolation from changes from the environment employs the use of feedback and calibration. Feedback is a continual process to remove short-term time varying processes, such as laser intensity at the laser head[3]. Calibration is used for slower processes, and those that determining parameters might be a large nonlinear process. Calibration is stored in a separate database, and is accessed directly by the user programs as named constants.

## 1.5 Design Philosophy

QuantumIon is designed around a multi-user, remote access idea. There is significant advanced hardware that is new, and innovative, from the standpoint of QuantumIon as purely an ion trap quantum computer: the use of $Ba^+$ ions has nearly all-visible wavelengths in its energy diagram, the use of an all-fibre scheme for individual addressing, improved vacuum techniques, and the use of the $Ba^+$ metastable states as 'shelving' states for non-binary quantum logic. However, this thesis focuses on the control system almost exclusively.

### 1.5.1 System High-Level Requirements

A remote-access system can be implemented in many ways, and this research adopts a typical best-practice systems engineering approach [17]. In particular, we take a Use Case

[2] In this case, *same* is from a statistical sense. A quantum computer is powerful because of the superposition of states in such experiments. Individual measurements of quantum systems can, and do, give different results.

[3] Intensity at the ion is adjustable, but requires a pre-stabilized source.

approach to this problem. To this end, the research aims to design a control system with three main goals:

- The precise control of classical electronic hardware to perform state preparation, measurement, quantum logic, readout, and stabilization;
- A consistent, flexible, powerful set of user operations (i.e. a user language) suitable for cutting-edge research;
- An IT environment that is robust, secure, scalable, and powerful enough to allow remote access via the Internet

### 1.5.2 User Descriptions & Requirements

In the use case methodology, it is paramount to perform a thorough survey of the types of different users that might potentially operate the system. The types of users envisioned are university-level graduate and professional research groups. Types of research considered are those who have a good quantity of published material related to ion-trap quantum information experiments. Because QuantumIon is an ion trap, other types of interesting quantum information theory not yet ready for experimentation, or those which lack ion trap protocols had to be discounted. For each type of researcher, if possible a typical experimental protocol was analyzed and used to inform the design of QuantumIon's capabilities.

As a result of this survey, QuantumIon is designed with the following users in mind

- Simulation Researcher - Research in this type of application centers around simulating a quantum system using an ion trap quantum computer, particularly analog simulation or hybrid simulation.
- Nonbinary Qudit Researcher - Research in this area centers around the specific properties of the $d > 2$ levels that the $Ba^+$ ion provides.
- Gate Optimization Researcher - Researchers in this area are expected to be studying the fundamental and systemic noise of the quantum computer and applying methods to improve system performance
- Quantum Error Correction Researcher - Research in this application centers around the implementation of quantum gates for the mitigation of decoherence and other loss of quantum state

- Cryptography Researcher - Research in this area involves quantum-safe cryptography. This researcher, while important in overall quantum computing field, is not expected to be a user of QuantumIon.

- Optimization Researcher - This type of research is involved in the optimization of complex systems, such as parameter optimization.

In addition, we identify two types of non-researcher, but whose interaction with the QuantumIon machine is of great importance

- Calibrator - This person, who may also be one of the researchers listed above, has special access to otherwise *dangerous* controls. The calibrator provides special quantum programs to define the machine's operating parameters. Regardless of their other responsibilities, in the Calibrator role he or she must be primarily concerned with quantifying the system, not with performing research.

- System Administrator - This person has the responsibility of creating new accounts, granting access, performing backups and diagnostics. In this role, the administrator is a fairly standard IT professional, who is familiar with the operation of databases, computer networks, and IT security.

## 1.6 Ion Trap Quantum Computers

An ion trap quantum computer has a fairly standard design process as described in literature [18],[19],[11]. The major components of the mechanical platform are: an ultra-high vacuum chamber, a Paul trap with RF drive, magnetic field coils, and optical access for the various lasers. A simplified representation is shown in Figure 1.2. In addition to the mechanical platform, a complex arrangement of optical components is necessary to create precise electric fields, and to deliver them to ions within the trap. The engineering of the optical components for control and beam delivery is a major effort in the design of any ion trap, and QuantumIon is no exception.

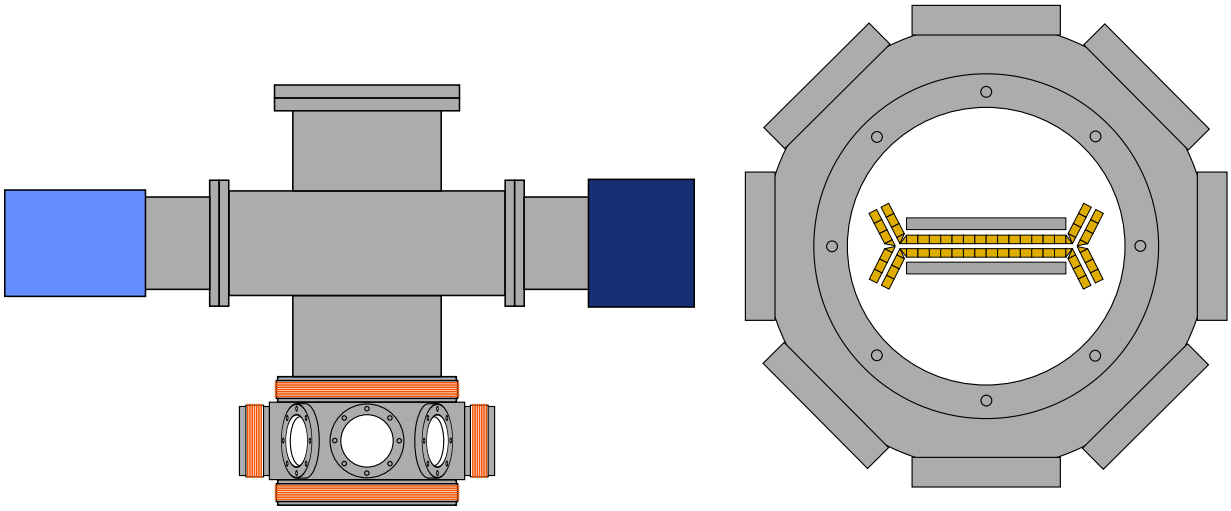

Figure 1.2: Ion Trap Chamber Side View (left) and Top View (right). The ion trap itself is located at the bottom of the apparatus, surrounded by the magnetic field coils used to split the hyperfine energy structure. From the top view, the characteristic dog-bone shape of the ion trap is visible, as are the two parallel RF electrodes. The quantum zone comprises the center of the trap

The purpose of this apparatus is to isolate, confine, and manipulate a chain of charged atoms (ions). If the machine is properly designed and calibrated, a string of ions lines up in a row, forming a one-dimensional crystal as shown in Figure 1.3. The confining potential of the Paul trap is such that the ions behave like a quantum simple harmonic oscillator. As such, the energy is quantized in phonons of energy $E = \hbar\omega_i$, where $\omega_i$ is the $i^{th}$ motional frequency.

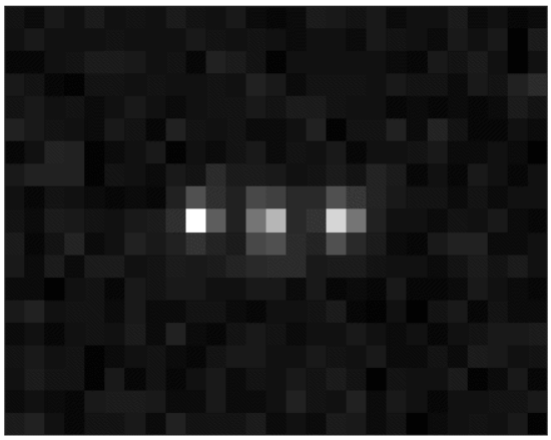

Figure 1.3: Three $Ba^+$ Ions in a 1-D Crystal. At Ultra-High Vacuum (UHV) pressures, the mean free path of background (untrapped) atoms is so long the ions are essentially isolated in space. In a Paul trap, Coulomb interaction forces the ions far enough apart that each atom can be spatially separated for manipulation and detection.

From a quantum information perspective, the ion trap quantum computer has two major features: the state of a specific qubit $|\psi\rangle$ is held in the electronic structure of the ion (i.e. the electron's state), and in the collective motional state of the ions as a trapped crystal[18]. The electronic structure is given in terms of the Pauli operator $\hat{\sigma}_z$, while the motional energy is given in terms of the phonon creation-annihilation operators $\hat{a}^\dagger$ and $\hat{a}$. For $N$ ions, the steady-state Hamiltonian is

$$\hat{H}_0 = \sum_{i=1}^{N} \frac{\hbar\omega_0}{2}\hat{\sigma}_z^{(i)} + \sum_{\nu=1}^{N} \hbar\omega_\nu \hat{a}_\nu^\dagger \hat{a}_\nu. \tag{1.1}$$

Under Equation 1.1, each ion is considered a 2-state system (a qubit) with an electronic transition $\omega_0$ with $\pm 1$ eigenvalues, and the crystal is a quantum harmonic oscillator with $\hat{n} = \hat{a}^\dagger\hat{a}$ phonons in the $\nu^{\text{th}}$ motional mode. To use computer design terminology, the electronic structure forms the data of the computer, while the crystal motional modes form the communications bus between the qubits.

### 1.6.1 Paul Traps

Charged particles interact with electric fields as described by Maxwell's equations. Through ionization, neutral atoms can be stripped of an electron, giving them the properties of a

charged particle. However, the Gauss law for electric field $\mathbf{E}$, and associated Laplace equation for its potential $U$, $\nabla \cdot \mathbf{E} = -\nabla^2 U = 0$ has no local minimum. This result, known as Earnshaw's Theorem, implies that particles cannot be confined in space by static fields alone. However, a time-varying electric field can, in fact, confine charged particles (and hence ionized atoms) in space.

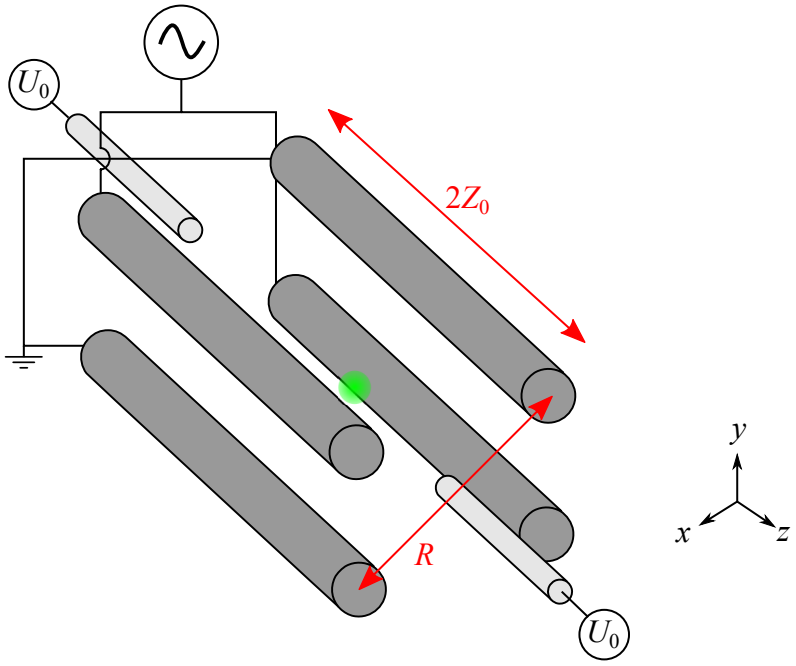


Figure 1.4: Four-Rod Paul Trap. This is among the simplest ion traps. An ion is confined in the center by static electric field potentials $U_0$ at each end, and RF potentials $V(t)$ at each of four corners. The resulting psuedo-potential is approximately a parabolic shape. The time-varying RF field ensures the ion stays at the center, with only a small micro-motion about the center.

The Paul trap is an arrangement of electric fields generated by conducting electrodes which carry RF and static DC fields as shown in Figure 1.4. For an infinite trap with potential $V(x, y, t)$ at every point along its $z$-axis, the RF portion of the potential, for a drive frequency $\Omega$, takes the form[20]

$$V(x, y, t) = \frac{V_0}{2}\left(1 + \frac{x^2 - y^2}{R^2}\right)\cos(\Omega t). \tag{1.2}$$

A typical arrangement, known as the *four-rod trap*, shown in Figure 1.4, contains four electrodes of length $2Z_0$, separated by diameter $R$, and two end electrodes at the ends with an applied DC voltage $U_0$. Near the center of the trap, these end electrodes contribute a potential

$$U(x,y,z) = \frac{\kappa U_0}{Z_0^2}\left[z^2 - \frac{1}{2}(x^2+y^2)\right]. \tag{1.3}$$

The total field experienced by an ion due by the applied electric field, $E = \nabla(U+V)$ is

$$E(x,y,z,t) = U(x,y,z) + V(x,y,t), \tag{1.4}$$

$$= -V_0\left(\frac{x-y}{R^2}\cos(\Omega t)\right) - \frac{\kappa U_0}{Z_0^2}(2z-x-y). \tag{1.5}$$

The motion of a single ion of mass $m$ and charge $Q$ in such a trap can be described by the one-dimensional Newtonian force in classical dynamics,

$$F = m\ddot{x} = QE(x,y,z,t), \tag{1.6}$$

$$\ddot{x} = -\frac{Q}{m}\frac{V_0}{R^2}(x-y)\cos(\Omega t) - \frac{Q}{m}\frac{\kappa U_0}{Z_0^2}(2z-x-y). \tag{1.7}$$

The $x$,$y$,$z$ versions of this equation can be rewritten in terms of the Mathieu equation,

$$\ddot{u_i} + (a_i + 2q_i\cos 2t)\frac{\Omega^2}{4}u_i = 0, \tag{1.8}$$

where the coefficients $a_i$ and $q_i$ are given as

$$a_x = a_y = -\frac{1}{2}a_z = -\frac{Q}{m}\frac{4\kappa U_0}{\Omega^2 Z_0^2}, \tag{1.9}$$

$$q_x = -q_y = \frac{Q}{m}\frac{2V_0}{\Omega^2 R^2}, \tag{1.10}$$

$$q_z = 0. \tag{1.11}$$

Solutions to Equation 1.8 define several stability regions along curves of specific values of $a_i$ and $q_i$. That is, only specific combinations of trap dimensions $R$ and $Z_0$, voltage amplitudes $U_0$ and $V_0$, drive frequency $\Omega$, and charge-to-mass ratio $Q/m$ will create stable

ion motion. In this case, unstable motion leads to ions leaving the trap (generally on trajectories that will stick to the surface of the containing vacuum chamber). Paul traps have the property that these stability curves describe regions of valid $Z$, $R$, $U_0$, $V_0$, and $\Omega$, so that the amplitudes may be tuned to select only a specific charge-to-mass ratio. Thus, the Paul trap not only confines individual ions in space, but also selects specific elements based on their mass.

The four-rod trap shown in Figure 1.4 is a simple example for analysis. In practical quantum computer experiments, electrode arrangement is replaced with $k$ sections of co-linear electrodes, each with a different value of RF voltage $V_k$ and DC voltage $U_k$. This arrangement yields more complex potential along the trap axis $z$, an essential requirement for confinement of more than one ion, and the creation of long ion chains. Examples of such traps include the 5-segment blade trap described by Debnath[11], and the HOA 2.0 trap used in QuantumIon[13] shown in Section 2.6.

### 1.6.2 Cooling, State Preparation & Measurement

The Paul trap described above is a device that can be designed, analyzed, and understood using classical electrodynamics. However, once the ion is contained within the Paul trap, the quantum state must be prepared. The particular method used to prepare the state is dependent on the atomic species chosen, but the general techniques are described in [21]. As an example[4], we consider singly-ionized Barium $^{133}Ba^{+}$. The energy level diagram is shown in Figure 1.5 [22]. Photoionization removes one electron from the $6s$ shell, creating a hydrogen-like atom. The state is prepared by cooling to the electronic $|6S, F = 1\rangle$ ground state using the $6S_{1/2} \rightarrow 6P_{1/2}$ transition at 493nm. Electrons may be caught in the $5D_{3/2}$ state due to spontaneous emission, and could be stuck in such a long-lived state; therefore a cooling repump transition is stimulated using the $5D_{3/2} \rightarrow 6P_{1/2}$ line at 649nm.

[4]The QuantumIon control system is agnostic to the atomic species. Suitable candidates also include $^{171}Yb^{+}$.

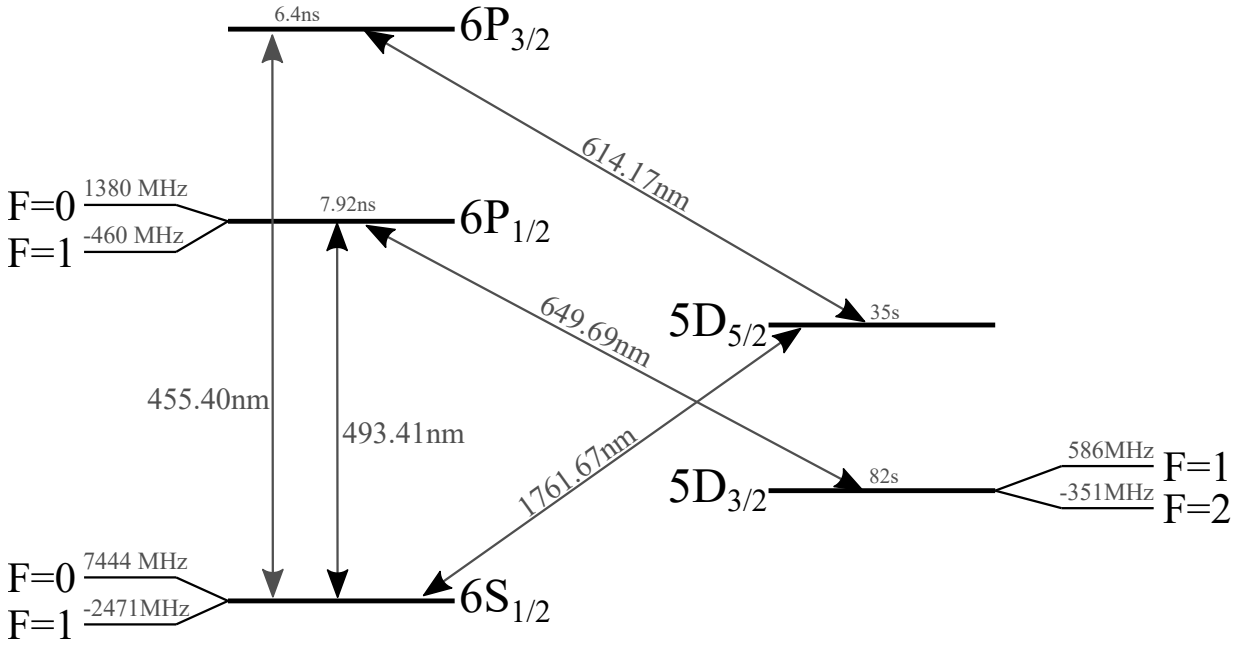


Figure 1.5: $^{133}Ba^{+}$ Energy Levels. The $6S_{1/2}$ and $6P_{1/2}$ manifolds provide the qubit computational states. The 493nm transition provides cooling and detection of the ion state. The long-lived $5D$ states could result in undesirable dark states, so repumping beams ate 614nm and 649nm are required. The 1762nm transition can be used as a so-called *shelving* transition to transfer population in the $6S$ manifold for measurement of $d > 2$-level systems.

Measurement of a quantum state takes on a similar approach. The detection beam at 493nm is detuned to the $|6S_{1/2}, F = 1\rangle \leftrightarrow |6P_{1/2}, F = 0\rangle$ transition. Since the corresponding $F = 0 \leftrightarrow F = 0$ transition is forbidden, any fluorescence detected is assured to be from the $F = 1$ state only, with appropriate use of the 649nm repumping beam.

### 1.6.3 Single Qubit Gates

The electronic states, such as $|6S, F = 1\rangle$, provide the physical representation of quantum information, however it is often more clear to speak in terms of Computational Bases. Such states can be mapped as $|0\rangle = |6S, F = 0\rangle$, and $|1\rangle = |6S, F = 1\rangle$.

For a pure state, a single two-level qubit can be represented as a linear superposition of states $|\psi\rangle = c_0 |0\rangle + c_1 |1\rangle$. The complex amplitudes must sum to unit magnitude. A

qubit can evolve in time according to a unitary operator $U(t)$ such that

$$|\psi(t)\rangle = U(t)\,|\psi(0)\rangle\,. \tag{1.12}$$

The operator $U(t)$ is a $2\times 2$ matrix, and can be expressed in the Pauli Bases $\{I, \hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z\}$, where

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \hat{\sigma}_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \hat{\sigma}_y = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix} \quad \hat{\sigma}_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1.13}$$

.

The Pauli basis provides an intuitive mapping between the matrix representation and the visual representation of Figure 1.6. The $\hat{\sigma}_x$ operator, for example, exchanges the amplitudes of the $|1\rangle$ and $|0\rangle$ states, which corresponds to rotation by $\pi$ about the $x$-axis. Similarly, the $y$- and $z$-axes rotations can be represented by corresponding Pauli operators. Since the Pauli matrices span the space of all $2 \times 2$ matrices, any unitary evolution $U(t)$ can be represented as a weighted combination of them, and therefore any (qubit) evolution $U(t)$ can be considered as rotations in $x$, $y$, and $z$.

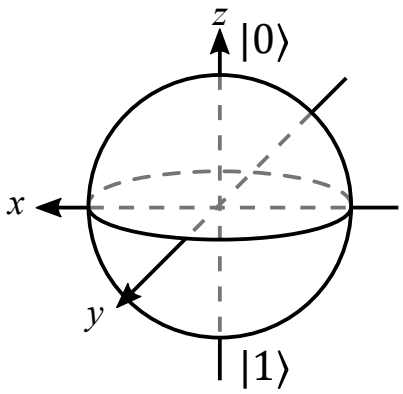


Figure 1.6: Bloch Sphere for Qubit. Population transfer between computational states $|0\rangle$ and $|1\rangle$ can be visualized as Pauli operators on the surface of a sphere. The $\sigma_x$, $\sigma_y$, and $\sigma_z$ operators have natural analogs to precisely-timed electric field pulses applied to the ion.

In an ion trap quantum computer, each ion is subjected to a beam of directed microwave, or laser, energy; i.e. an electromagnetic field. The description of how these rotations are implemented begins with a description of the ion interacting with an oscillating electric

field,

$$\mathbf{E} = E_0 \cos(\omega t + \phi). \tag{1.14}$$

For wavelengths that are large compared with the atomic radius, the ion can be considered as an electric dipole $\mathbf{d}$. In the Rotating-Wave Approximation (RWA), the interaction of the atom and the field has the interaction Hamiltonian [23],

$$\begin{aligned} H_{AF} &= -\mathbf{d} \cdot \mathbf{E}, \\ &= -\langle 0|\hat{\varepsilon} \cdot \mathbf{d}|1\rangle \left(E_0^+ \sigma e^{i\omega t+\phi} + \text{H.C.}\right), \\ &= \frac{\hbar\Omega}{2}\left(\hat{\sigma} e^{i\omega t+\phi} + \hat{\sigma}^\dagger e^{-i\omega t-\phi}\right), \end{aligned} \tag{1.15}$$

where the energy level lowering and raising operators are $\hat{\sigma} = |0\rangle\langle 1|$ and $\hat{\sigma}^\dagger = |1\rangle\langle 0|$, respectively. A major parameter of Equation 1.15 is the Rabi frequency, $\Omega = 2\langle 0|\hat{\varepsilon} \cdot \mathbf{d}|1\rangle E_0^{(+)}/\hbar$. Applying Schrödinger's equation in the rotating frame gives the dynamic equations for state $|\psi\rangle = c_0|0\rangle + c_1|1\rangle$

$$\begin{aligned} \dot{c}_0 &= -i\frac{\Omega}{2} e^{-i(\delta t-\phi)} c_1 \\ \dot{c}_1 &= -i\frac{\Omega}{2} e^{i(\delta t+\phi)} c_0. \end{aligned} \tag{1.16}$$

When on resonance, the detuning $\delta$ is zero, and the standard solution to Equation 1.16 is an oscillation

$$|\psi\rangle = \cos\left(\frac{\Omega}{2}t\right)|0\rangle - ie^{-i\phi}\sin\left(\frac{\Omega}{2}t\right)|1\rangle. \tag{1.17}$$

Equation 1.17 has the basic characteristic of a cyclic population transfer with frequency $\Omega/2$, plus an additional phase term $\phi$. As a result, the on-resonance electric field performs so-called Rabi Flopping between the states $|0\rangle$ and $|1\rangle$, and the *Rabi period* $\Omega^{-1}$ is the time it takes for a complete transfer cycle. As a unitary evolution $|\Psi(t)\rangle = U(t)|\Psi(0)\rangle$, the on-resonance operator is

$$U(t) = \begin{pmatrix} \cos\frac{\Omega}{2}t & -ie^{-i\phi}\sin\frac{\Omega}{2}t \\ -ie^{i\phi}\sin\frac{\Omega}{2}t & \cos\frac{\Omega}{2}t \end{pmatrix}. \tag{1.18}$$

This unitary evolution provides a population swap when $\frac{\Omega}{2}t = \pi$ and $\phi = \frac{\pi}{2}$, equivalent

to the Pauli $\hat{\sigma}_x$ operator. Similarly the Pauli $i \cdot \hat{\sigma}_y$ operator occurs when $\frac{\Omega}{2}t = \frac{\pi}{2}$ and $\phi = \frac{\pi}{2}$. A useful intermediate operator, the $\sqrt{\hat{\sigma}_x}$ operator[24], provides a superposition state $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ when $\frac{\Omega}{2}t = \frac{\pi}{2}$ and $\phi = 0$. With this operator, the Pauli $\hat{\sigma}_z$ operator is the composite sequence $\sqrt{\hat{\sigma}_x}\hat{\sigma}_y\sqrt{\hat{\sigma}_x}$.

The above results show that the basic Pauli gates[5] can be constructed from a unitary evolution $U(t)$, which depends only on two physical parameters: the pulse duration $\frac{\Omega}{2}t$ and the phase $\phi$. These controls are realized physically by the phase of the drive signal that modulates the optical beam and relative time between pulses, giving $\phi$, and the pulse duration $t$ for a given Rabi frequency. Further, since the Rabi frequency is dependent on the dipole moment, and hence the electric field amplitude, control of the field is a useful control for optimization.

In QuantumIon, the electric field transition frequency is in the microwave region of several GHz, and is generated by beat notes between two Raman lasers, but direct control of the electric field by a microwave generator is also possible.

### 1.6.4 Entangling Gates

The state of Quantum Entanglement is among the most uniquely non-classical features of a quantum system. A pair of qubits are entangled if they are in a state which is not a tensor product of any two individual states, that is

$$|\psi\rangle \neq |\psi_1\rangle \otimes |\psi_2\rangle \,. \tag{1.19}$$

A common form of entangled state is the so-called Greenberger-Horne-Zeilinger (GHZ) states[25], which are the superpositions of repeated, identical eigenstates. In the spin-$\frac{1}{2}$ systems common in atomic physics, these are the all-up and all-down states, $(|\uparrow\uparrow .. \uparrow\rangle + |\downarrow\downarrow .. \downarrow\rangle)/\sqrt{2}$.

A robust two-ion entangling gate is the Mølmer-Sørensen gate (MS gate)[26]. In this scheme, population from a two-qubit state $|\downarrow\downarrow\rangle$ can be transferred to a state $|\uparrow\uparrow\rangle$. Since a direct transfer $|\downarrow\downarrow\rangle \leftrightarrow |\uparrow\uparrow\rangle$ is forbidden, the transfer is performed using intermediate states $|\uparrow\downarrow\rangle$ and $|\downarrow\uparrow\rangle$.

The chain of ions in a linear trap forms a Quantum Harmonic Oscillator (QHO)[8], which form by the motional modes from Coulomb interaction between the ions. The

[5]This discussion focuses on the Pauli formalism common in quantum information texts[4]; a more natural approach for the ion trap physics is in the form of rotation operators[24].

intermediate ion electronic states, $|\uparrow\downarrow\rangle$ and $|\downarrow\uparrow\rangle$ are split by equally spaced motional modes of a quantum harmonic oscillator. The energy levels are quantized by the phonon number $n$, as shown in Figure 1.7.

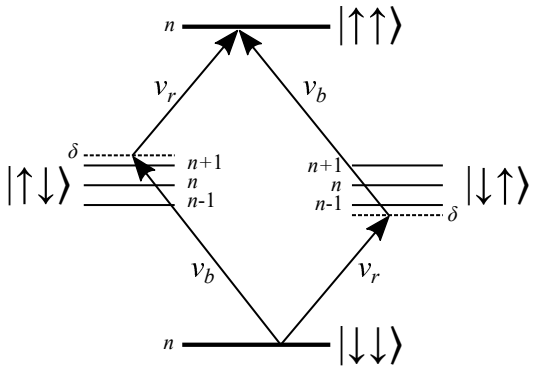


Figure 1.7: Mølmer-Sørensen gate in energy space. Figure based on [27]. The entangled states $|\downarrow\downarrow\rangle$ and $|\uparrow\uparrow\rangle$ are traversed by the intermediate states $|\downarrow\uparrow\rangle$ and $|\downarrow\uparrow\rangle$ via laser beams corresponding to adding and subtracting one phonon of motional energy to the ion chain respectively. The resulting beams frequencies are red and blue motional sidebands of this intermediate transition. Symmetry ensures only two sidebands are required, and that the transition is relatively insensitive to the starting motional state $|n\rangle$. The MS gate is therfore attractive since it does not require cooling to the motional ground state $|n = 0\rangle$.

A properly applied laser can access the electronic states, and the motional (phonon number) states of an ion chain. The transition frequency between $|\downarrow\downarrow\rangle\,|n\rangle$ and an intermediate state with one less phonon, $|\downarrow\uparrow\rangle\,|n-1\rangle$, can be accessed through an electric field of frequency $\nu_r$ that has been detuned by $\delta$. The population transfer can be completed by a frequency $\nu_b$ between $|\uparrow\uparrow\rangle\,|n\rangle$ and $|\downarrow\uparrow\rangle\,|n-1\rangle$, similarly detuned by $\delta$.

The MS gate takes advantage of the symmetry between the upper (blue), and lower (red) sideband given by the equally spaced motional modes, such that $\nu_r$ can access $|\downarrow\uparrow\rangle\,|n-1\rangle \leftrightarrow |\uparrow\uparrow\rangle\,|n\rangle$ and $|\uparrow\downarrow\rangle\,|n+1\rangle \leftrightarrow |\uparrow\uparrow\rangle\,|n\rangle$. Due to the equal spacing of the QHO motional mode frequencies, this gate is not sensitive the actual phonon number $n$, and so the MS gate does not require the difficult cooling to $n = 0$ that is required by the original Cirac-Zoller gate[28].

## 1.7 Role of the Control System

The physics described above to provide single-qubit gates, and entangling gates, require a sophisticated control system.

Cooling operations, for example, require a specific wavelength of radiation to be delivered to the ion. The precise wavelength is provided by modulating the output of a diode laser. Modulation is provided through the application of RF signals to Acousto-Optic Modulator (AOM) and Electro-Optic Modulator (EOM) modules. Therefore, the control system must provide such RF signals.

Single-qubit operations require similar wavelength tuning, with the additional requirement of precisely timed pulses; the control system must therefore provide a means of precisely turning on and off these beams[6].

Entangling operations require the generation of multiple tones for the red and blue motional sidebands, and best practices also require control of the exact phase, frequency, and amplitude profile of these pulses.

The readout of a quantum state comes in two forms: either direct measurement of the ion by imaging camera, or by detection of photons using a Photomultiplier Tube (PMT). As a result, the control system must be capable of collecting counts, and of collecting (and processing) images.

Automation of the optical system is achieved through a combination of physical manipulators and feedback controllers. Polarization control of beams is achieved by rotating waveplates, and associated motor controls. Beam positioning is achieved by piezeoelectric transducers on mirror mounts, coupled with two-dimensional sensors for feedback.

There are also support electronics under the control of QuantumIon. These include laser diode controllers, calibration wavemeters, and various environmental and Heating, Ventilation, and Air-Conditioning (HVAC) controls. These interface to the control system using standard serial port connections, such as RS-232 and RS-485, and via Ethernet.

These examples illustrate the wide range of controllers needed to operate QuantumIon. One of the core ideas of the control system is to provide a homogeneous view of the apparatus. As a result, the control system must interface with these different systems and provide a means for user control (where possible), and background stabilization, calibration, and monitoring.

[6] Again, the AOM and EOM provide this

## 1.8 Conclusion

This chapter introduces the problems of basic quantum computing using trapped ions, and to the QuantumIon project. It can be seen that the ion trap provides a suitable platform for the exploration of the unique properties of quantum computation, through the unique physics of a chain of ionized atoms that are confined in space. Under these conditions, and with suitable isolation from other stray matter and fields as provided by an ultra-high vacuum chamber, the very interesting quantum properties emerge in a way that can be manipulated with precisely controlled lasers.

Due to the specialized nature of the apparatus, and the expense of equipment needed, ion trap quantum computers have been prohibitively expensive and too specialized for many types of research. The QuantumIon project aims to bring this powerful platform to a wider audience, in the hopes to spread the cost and increase usage.

The technical details of the system begin with the next chapter. In it, the specific focus of this thesis are described at the overall architecture level.

# Chapter 2

# System Architecture

The previous chapter introduced the QuantumIon system, and the basic physics of ion trap quantum computers. In this chapter, the mechanical system, the ion trap itself, the optical system, and the computer control system will be described. This *architecture* level description of QuantumIon aligns naturally with the major efforts of the actual division of labor amongst the members of the QuantumIon design team. This thesis is mostly concerned with the control system, which represents the bulk of the author's work; the extraordinary efforts of the mechanical team, ion trap integration team, and optical engineering team should be acknowledged, and it is important to understand how these parts fit together.

In its broad term, the system architecture is the collection of high-level technical ideas that compose the end solution. In this chapter, these high-level concepts are explored to show how the user requirements and functional requirements are satisfied. It is the goal of a good system architecture that these requirements are satisfied consistently, robustly, and completely.

## 2.1 Ion Trap

The ion trap chosen is the Sandia National Lab High Optical Access (HOA) 2.0 surface trap[13]. This pre-fabricated trap provides a series of electrodes in three major areas: the primary quantum area, in the center of the double-Y shape, four loading zones at the extreme ends, and a transition zone at each juncture. Additionally, the HOA trap provides a separate set of RF electrodes for the confinement along the trap axis.

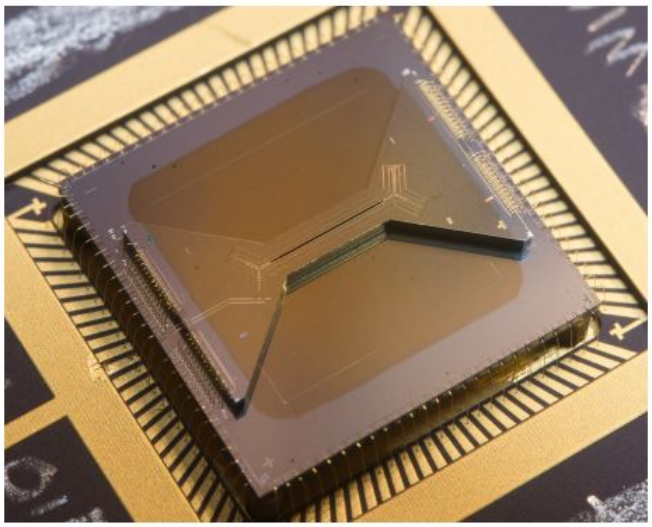

Figure 2.1: High Optical Access Trap used in QuantumIon (picture from [13]). This ion trap is produced by Sandia National Laboratories for ion trap research. The trap fits on a 1 × 1-inch chip carrier. Approximately 92 electrodes line the central slot and side arms. The characteristic *dog-bone* shape comprises a central quantum zone, four loading zones and the RF drive electrodes.

## 2.2 Optical Architecture

Once trapped, the primary means for manipulating quantum states is via optical fields. Several distinct beams and different wavelengths are required for proper operation. The physical orientation is shown in Figure 2.2. The central trapping region is partitioned into two zones: a loading zone off-center, and a central quantum manipulation zone.

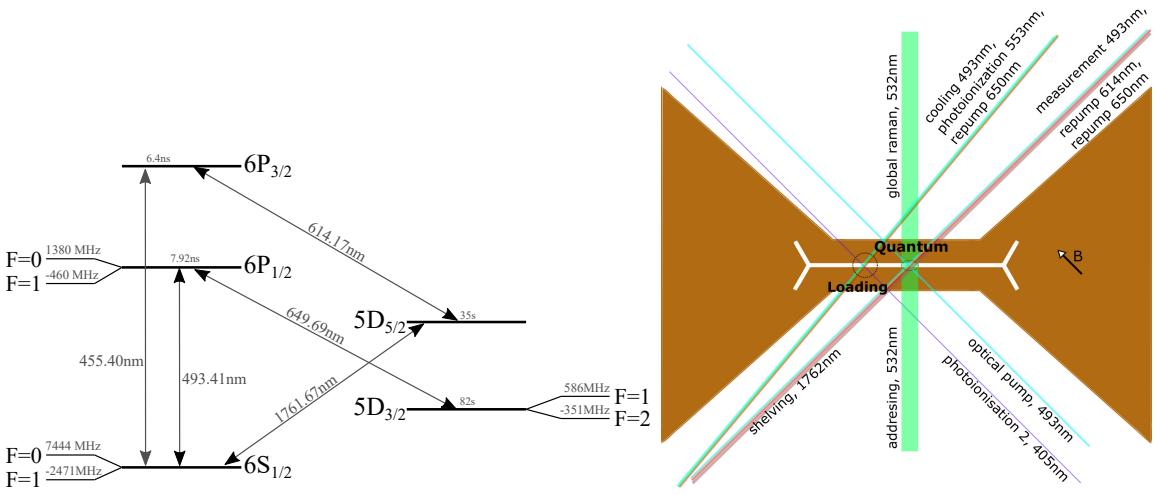


Figure 2.2: $^{133}Ba^{+}$ Energy Levels & Optical Beam Delivery Paths. The ion trap (right) provides two physical areas for optical focus: a *loading zone*, where neutral atoms are ionized, forming charged particles that can be trapped; and a *quantum zone*, where qubit manipulations take place. Ions are shifted from the loading area to the quantum area via shuttling by time-varying electrode voltages. Note cooling and repump are required at both sites for effective trapping, while only the quantum zone requires the shelving and Raman beams.

Beginning with a neutral atoms liberated from the source material by photoionization or Joule heating, atoms are ionized using a two-photon photoionization process[29], with lasers at 553nm and 405nm. Cooling the ions to near the ground state, $^{6}S_{1/2}$, is accomplished via the 493nm beam, which also excites the fluorescence for state measurement. The primary quantum state manipulation occur along two counter-propagating Raman beams at 532nm. The Raman beams consist of a single global beam that is focused on all ions simultaneously, and individually addressing beams which are each focused on a single ion. During cooling, some ions may be stuck in a long-lived dark state at $^{5}D_{3/2}$, and so a re-pump beam at 649nm transfers population back to the main cooling transition $^{6}P_{1/2} \leftrightarrow ^{6}S_{1/2}$. Finally, for access to the metastable *shelving* state at $^{5}D_{3/2}$, the 1760nm beam drives this transition. Similar to the dark state problem for cooling, a shelving repump laser at 614nm drives the transition out of the long-lived $5^{D}_{5/2}$ state and into the short-lived $^{6}P_{3/2}$.

## 2.3 Optical System Controls

QuantumIon's optical system relies heavily on automation and electronic control to provide remote access. After initial set-up, most manual adjustments are controlled by digital control loops, or periodic calibration with electronic control. Figure 2.3 shows a typical optical schematic for the Cooling/Repump/Measurement beam at 493nm. The laser head itself is controlled by serial commands based on feedback from a commercial wavemeter. Polarization control and cleanup are provided by rotating motor controls with optical intensity feedback. Sideband control is provided through an RF signal provided to an EOM, using a Fabry-Perot etalon as an optical filter and sensor. Fine-grained wavelength tuning is provided by RF control to an AOM. Precision beam pointing and alignment stabilization is provided by piezoelectric transducers on mirror mounts, using two-axis sensors and imaging cameras. Finally, beam quality and alignment are assisted by a series of Charge-Coupled Device (CCD) cameras and under real-time image processing control.

This sophisticated optical set-up, and the associated electronic controls and software, are necessary for the automation goals of QuantumIon. Each sensor's feedback and control parameters are logged over time, and this provides a great deal of flexibility in the machine's operation.

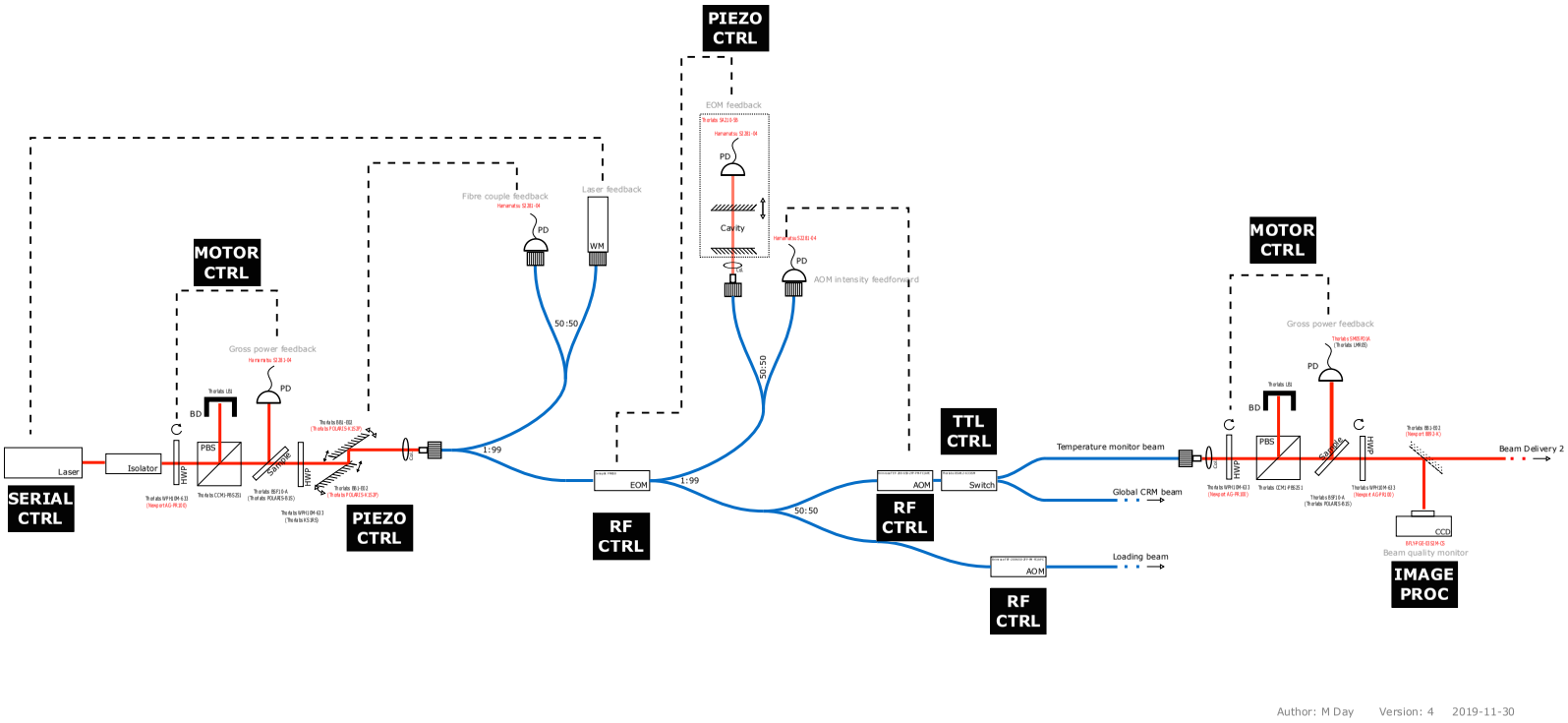


Figure 2.3: Typical Optical System Schematic. Traditional optical components, such as lasers, waveplates, and modulators are under computer control. The use of a series of feedback control loops provides stability, and the control of these feedback setpoints provides automation and programmability. (Schematic by Matt Day, QuantumIon design team)

## 2.4 Vacuum System

The vacuum system is the primary mechanical system of QuantumIon. In addition to providing mechanical support, the vacuum system's main purpose is the establishment of an UHV environment. The vacuum is required to limit the destruction of the ion chain, or decoherence of quantum state, by collisions with free gas molecules. The CAD model of the vacuum chamber is shown in Figure 2.4. The details of the vacuum system will be documented in the upcoming thesis by Noah Greenberg of the QuantumIon group.

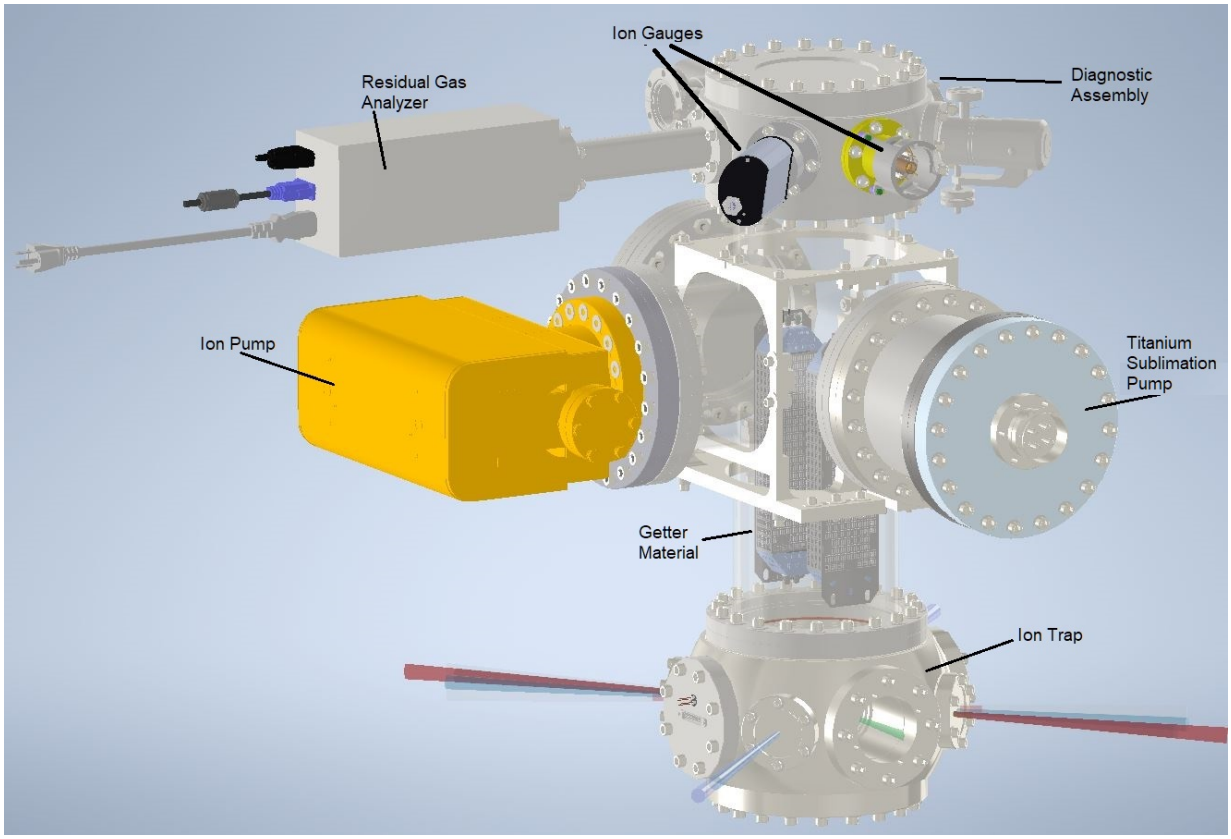


Figure 2.4: CAD Rendering of Vacuum System. The diagnostic assembly and Residual Gas Analyzer (RGA) form the upper portion. The ion pump, and getters and ion gauge provide and monitor the ultra high vacuum. The ion trap and associated laser beams form the bottom of the physical assembly. The structure is supported on all sides for mechanical stability (not shown).

The UHV is established by first a bake-out of remaining hydrogen and water that is impregnated in the metal structures. Prior to bake-out, the internals are assembled (including the surface trap and associated wiring). The chamber is sealed with copper gaskets, and the atmosphere is *rough pumped* using a conventional turbo-molecular pump. The thermal limits of optical viewports and the chip trap limit bake-out to approximately 150°C for one week. From this point, the chamber environment is sealed, and an internal ion pump is used to provide and maintain the extremely low pressures required by the quantum experiment.

The main chamber, housing the trap itself, is shown at the bottom of Figure 2.4. The central region contains the ion pump, getter material for hydrocarbon absorption, and titanium sublimation pump as a backup absorber. Finally, the top area contains the diagnostic subassembly, housing the ion gauges for pressure measurement, and RGA.

## 2.5 Computing Hardware Architecture

The computing hardware represents the bulk of the work performed in this thesis.

The high-level computing architecture is broken up into a few major pieces as shown in Figure 2.5. Except for the user program, all components reside in the datacenter of the QuantumIon system. The major components are as follows:

- The *User Program*, which resides on the users computer (a laptop or desktop at a remote office). This program is written in one of the binding languages (e.g. Python, Matlab, etc). These Language Bindings are described in detail in Chapter 6.
- The *Main Program*, which provides all sequencing, security, user access and coordination for the QuantumIon system
- A set of *Linux Drivers*, which interface the main program to the low-level hardware.
- A set of *FPGA Cards*, which reside on special high-speed PCI Express backplane chassis and provide the timing-critical logic needed for all core processing operations. These are detailed in Chapter 3.
- A set of *FPGA Mezzanine Card (FMC)*, one connected to each FPGA card, to provide a specific type of electrical interface (e.g. Analog-to-digital conversion at some sampling rate).

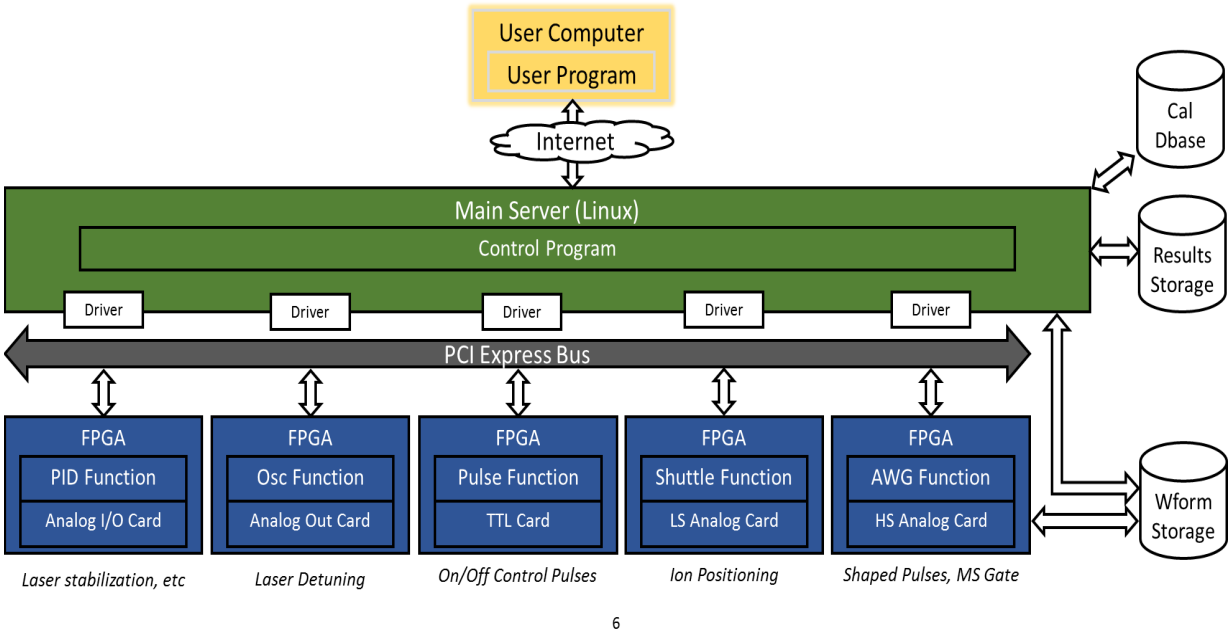


Figure 2.5: High-level computing architecture. The user connects via the internet to the main server and control program. User programs are decoded and the latest calibration values are applied to the final program. Custom Linux drivers allow the main program to access the individual FPGA modules, which perform the precision-timed functions used to control QuantumIon. FPGAs also capture measurement results and store them in semi-permanent storage after the experiment is over. Additionally, a dedicated high-speed network is used to play back custom waveforms for pulse shaping and control.

- A set of *FPGA Functions*, which is code written in the FPGA Hardware Description Language (HDL) and implement functions such as TTL output, PID control, frequency generation, and so on. These are detailed in Chapter 3.
- A *Calibration Database*, which stores the latest calibration settings for derived parameters. This is discussed in detail in Chapter 8.
- A *Results Storage Array*, which stores the results of measurements for a short period between when a quantum program is run and when the user downloads that data to their local computer.
- A *Waveform Storage Array*, which stores the AWG patterns provided by the user for access by the AWG hardware. AWG hardware is discussed in Chapter 5.

## 2.6 Software Architecture

The software architecture is described in detail throughout this thesis. From a user-centric point of view, the main elements are shown in Figure 2.6, which describes the high-level evolution of a user program.

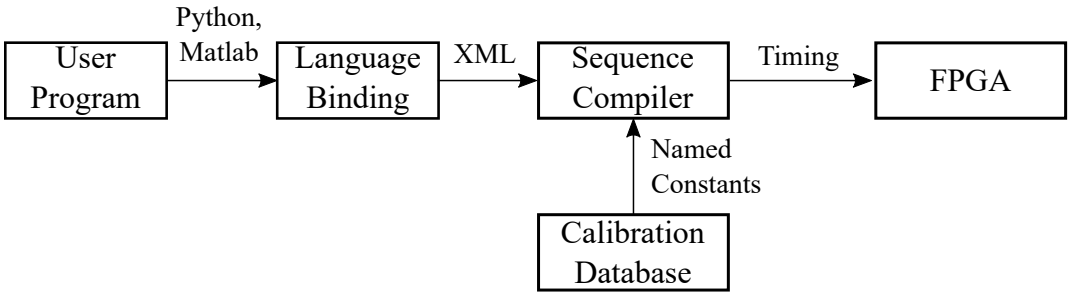


Figure 2.6: User program evolution. User programs begin at the user computer in a suitable high-level scientific programming language. The language is converted to XML before being transmitted to the QuantumIon server. On the server, when the user's program is selected for execution, the sequence compiler creates the final timing commands using the latest calibration database, and sends these commands to the FPGA for execution.

The user program is generated by a remote user on his or her computer. The user program computer is important, in that it is outside the control of QuantumIon but has

very specific responsibilities for the overall successful use of QuantumIon. The user creates quantum programs in their favorite programming language. A library, called a Language Bindings, converts this programming language, such as Python or Matlab, into an intermediate eXtensible Markup Language (XML) language. This XML is sent to the QuantumIon server, where the sequence compiler, using information from the calibration database convert this program into a series of precisely-timed operations for the FPGA processors. Results are returned to the user's computer, where post-processing can occur.

From this, the highest-level view of the QuantumIon control system, the user's computer is used to generate programs, the QuantumIon server executes these programs, collecting data along the way, and then the user's computer downloads and interprets these results. The separation is intentional for two reasons. First, the requirement of multiple users is met by ensuring the QuantumIon machine becomes ready for the next user as soon as the last user's data is collected; post processing is a classical computing problem, while machines that can perform quantum operations are rare indeed. Secondly, post-processing tools evolve rapidly from commercial products and open-source communities; to keep up with the plethora of such tools would present a challenge to maintainers of QuantumIon's code base.

## 2.7 FPGA Architecture

The FPGA architecture provides precise timing of most physical electrical controls. These electrical controls include Transistor-Transistor Logic (TTL) outputs (for precise triggers), TTL inputs for counter-type measurements, Direct Digital Synthesizer (DDS) for optical modulators, Arbitrary Waveform Generator (AWG) for ion manipulation, and image processing for CCD cameras.

Since successful FPGA implementation is critical to the success of the QuantumIon machine, a brief primer is included in Appendix C. The detailed FPGA architecture of QuantumIon is described in detail in Chapter 3.

In the QuantumIon control system, each FPGA is an autonomous entity to ensure precision, high-speed operation. Each FPGA receives a set of instructions for the quantum experiment directly from the server PC prior to the experiment start. Synchronization between cards is of the utmost importance and so all devices share a common clock and trigger. Once the program is completed, each FPGA that is capable of performing measurements uploads its results back to the main server for storage, and later, retrieval by the user.

## 2.8 Conclusion

In this chapter the highest-level view of the QuantumIon control system is described. Particularly, the system architecture ties together three major components: the mechanical design exemplified by the vacuum system, the ion trap and optics system, and the computer control system. The control system can also be subdivided into the specialized FPGA modules and their programming, and the *server* infrastructure (composing all non-FPGA parts) and associated software.

The next chapter explores in depth the FPGA modules, their hardware, and software. It will be shown that these highly programmable devices are designed and programmed in a very different way from standard programming on a personal computer. Although FPGAs provide a very different type of computing, they are essential for the strict precision timing required for ion trap experiments.

# Chapter 3

# FPGA Hardware

In the previous chapter, the overall QuantumIon apparatus was broken into a series of major functional parts, mimicking the efforts of the members of the QuantumIon design team. It was shown that the control system requires heavy use of specialized programmable hardware. In this chapter, the details of these programmable devices are explored to achieve the precision needed. There will be considerable parallels with the design of the FPGA software, and the concepts of the main server program (Chapter 4) and the custom programming language (Chapter 6).

Field-Programmable Gate Array (FPGA) hardware provides all time-critical electronics control. All electronics that are part of the *quantum program* pass through the FPGA hardware. Although some functions are inherently low-speed (such as stabilization of magnetic field), all FPGA hardware includes an execution engine that operates at a single 2 GHz sample rate. For these low-speed devices, one can imagine the 2GHz rate as defining the resolution of the *start time* of any parameter change, while the *duration* may be on the order of milliseconds or longer. Examples of high-speed FPGA modules include AWG, TTL pulse, and DDS modules, while low-speed modules include temperature control, magnetic field PID loops, and shuttling.

The use of this single 2GHz timing reference provides a consistent design; those processes that operate at lower speeds are downsampled just prior to output.

## 3.1 FPGA Execution During a Quantum Program

The evolution of a quantum program, beginning with the user's source code, is described at several points in this thesis. For this chapter, is useful to describe this evolution from the perspective of the FPGA modules themselves.

After the user's xml program is compiled as described in Section 4.2, a series of Operation Codes (Opcodes) is generated for each execution engine (described below). These opcodes are uploaded to the various functional blocks on each FPGA. A sample clock has been distributed to each module, At this point, the QuantumIon FPGA system is ready for execution, and each module is operating independently[1].

A single trigger signal is generated to begin the execution. Throughout execution, the global time is checked, and when an opcode is scheduled for execution, the FPGA module makes the appropriate change. Many modules simply make changes to a parameter of a complex function (such as phase of a DDS generator). Some modules receive and calculate data, such as PMT counts, and this data is stored on the FPGA during the experiment. Shared data used in branching logic is broadcast to other FPGAs. Over the course of the experiment, the list of opcodes may run through loops and conditional logic in addition to making parameter changes. The server is notified at the end of the experiment, after which the FPGAs end independent execution and once again wait for instructions from the server. At this point, the server can extract the measurement data from the FPGAs and place it in storage for the user to request.

The idea that the FPGAs run independently, but synchronized to a single clock, is key to achieving sub-nanosecond execution. Should the FPGAs have to wait on the main server for instructions, unpredictable latency would result. It is for this reason that several existing commercial control systems could not be used in QuantumIon.

## 3.2 Basic Interconnect Scheme

Although FPGA modules implement a large number of functions, they have much in common with each other. Each FMC card, regardless of function, is programmed using consistent concepts: 2GHz experiment clock, experiment start trigger, and programming via PCI Express bus.

[1]Some actions, such as partial measurements, will broadcast to the other FPGAs, but the modules do not wait on each other, or the main server.

### 3.2.1 Sample Clock

All components in the QuantumIon control system are slaved to a single master clock: in this case, a 10 MHz Rubidium frequency standard. This clock is distributed to all FMC sites. Figure 3.1 shows the clocking layout. The master 10MHz reference is distributed to each FPGA device (AWG, DDS, shuttling Digital-to-Analog Converter (DAC), other stabilization). Within each FMC card is a Phase-Locked Loop (PLL), which upconverts the 10MHz reference into the 2GHz experiment clock. This scheme ensures a consistent timing reference even though each FPGA module must generate its own clock internally.

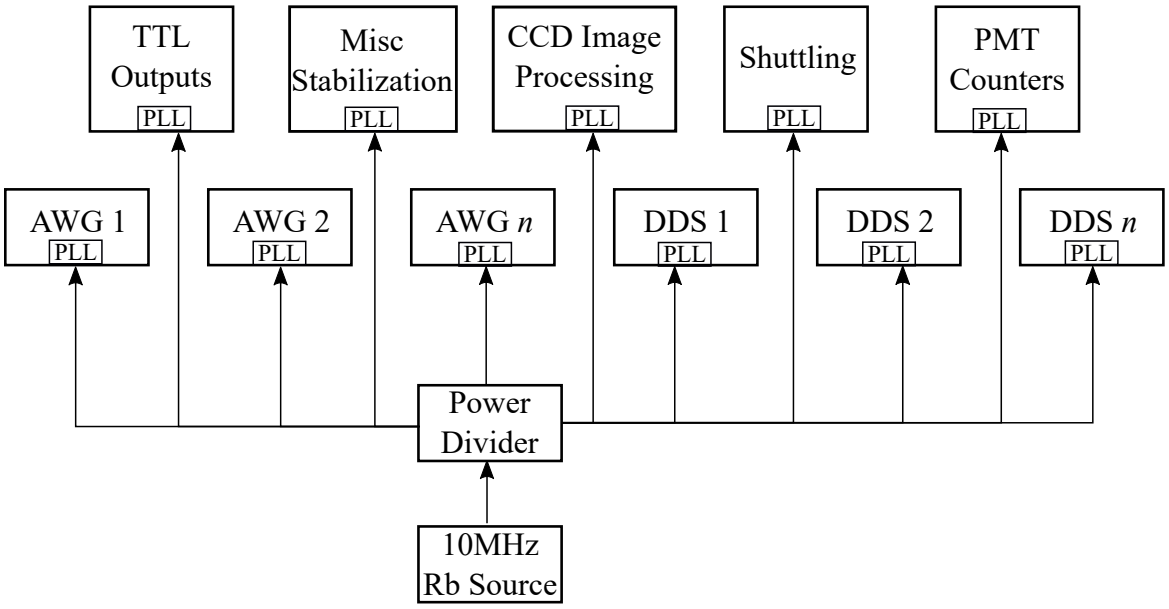


Figure 3.1: Sample Clock Distribution: A single atomic reference 10MHz clock is distributed via phase-matched cables to each FPGA. Within each FPGA is a PLL, which provides the internal 2GHz master clock for each module's timing function.

### 3.2.2 Experiment Start Trigger

The master 10MHz clock is locally upconverted to the experiment 2GHz clock within each FMC card. The phase locked loops within each card ensure that the 2GHz clock is on-frequency, and phase aligned, but there is still ambiguity about when the $t = 0$ epoch of any experiment starts. To that end, a single *start* signal is shared to each card from a common source, similar to how the 10MHz is distributed. A single pulse generator signals the start

of an experiment, and this occurs only after all FPGA cards have been programmed with the experiment instructions. The distribution network is shown in Figure 3.2.

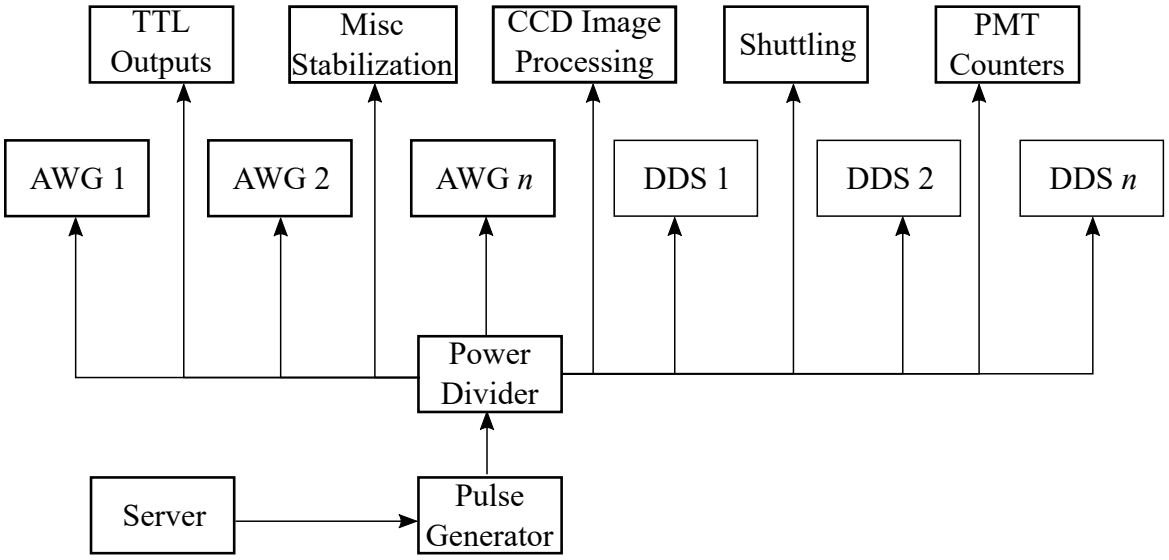


Figure 3.2: Experiment Trigger Distribution. A single trigger pulse is distributed via phase-matched cables to each FPGA module in QuantumIon. The execution engines in each FPGA are armed with the current quantum program timing information. Upon triggering, the execution engines operate separately until the experiment is over.

### 3.2.3 PCI Express Interconnect

All communication, configuration, and readback is performed through the PCI Express interconnect network. PCI express provides a backplane-style[2] network that directly connects the server to each FPGA. FPGA cards reside in a series of rack chassis in the electronics bays of QuantumIon. Each chassis contains a PCIe extension card that directly extends the server motherboard backplane. As a result, each FPGA card appears to the server as an appliance on its motherboard. The main program (see Chapter 4) interfaces with the FPGA card through a series of Linux custom drivers. These drivers map the configuration space of the FPGA card to driver system calls accessible from the main program.

[2]Backplane interconnects are familiar from personal computer motherboards, where the cards all connect to a single circuit board instead of individually wired back. In QuantumIon, the rack becomes a sort of second motherboard to the server.

### 3.2.4 InfiniBand Network

Although configuration of each FPGA and the resulting execution graph (see Section 4.4) are communicated through PCI express, this protocol is insufficient for the partial measurements needed for the hybrid classical-quantum programs needed for quantum error correction. In particular, the PCI express protocol is built around a *root complex* idea [30]. In this topology, communication between chassis must return to the root complex (located on the server).

Consider the following example: A partial measurement (such as a CCD measurement of fluorescing ions) is processed in an image processor FPGA, and the resulting measured state must be broadcast to the other cards. Such a measurement data packet must go to the chassis controller, return to the server root complex, and then travel to the destination chassis controller, then to the FPGA card of interest. Such latencies are likely to be unacceptable in large systems. The large system idea is very much a real concern as future experiments may require scalability of, e.g. 10 traps with 10 ions each [31].

InfiniBand is an alternative communication system to PCI express that alleviates the large-scale quantum processor in two ways. Firstly, as a switched fabric as opposed to a hierarchical network, a 1-hop path exists from any FPGA card to any other (the single hop is the InfiniBand switch). Secondly, the 10-by-10 system above consumes up to 100 FPGA cards, which can exceed the bus sizes of some PCIe implementations. InfiniBand networks scale to $2^{128}$ network nodes; a limitation that is unlikely to be exceeded.

To implement the InfiniBand network, each FPGA card is equipped with the necessary FPGA intellectual property (IP) core to communicate through the network. Partial measurements and other sundry data can be broadcast to all other FPGA cards directly through the InfiniBand switch. Similarly each FPGA contains an InfiniBand receiver core, allowing the new partial measurements to be used as soon as they are available.

It is expected that this topology will not be implemented in the first-generation Quantum-Ion control system. Instead the local PCI Express bus used to communicate with the measurement information, and Infiniband is planned as an upgrade.

### 3.2.5 Interpolation of Parameters

All parameters[3] highlighted in the sections below can be manipulated by the user over the course of the program. However, discontinuous jumps between one parameter and another

[3] An example of a parameter might be a DDS frequency, shuttling electrode voltage, or magnetic field setpoint.

may be disruptive to the physics of the program at hand. Similarly, a long sweep of a parameter such as the frequency shift required during rapid adiabatic passage, should be smooth. Synthesizing these steps is possible, but requires the user to program a large number of intermediate steps.

A better solution is to use *interpolation* so that the FPGA modules might fill-in the intermediate parameter values as the program executes. Several methods exist to provide such interpolation: zero-order hold, linear, polynomial, spline, etc. Although these fit under a general category of *function approximation*, a defining characteristic of the interpolation problem is the user provides the desired output at two points $\Theta_1$ and $\Theta_2$. From there, some interpolating function generates the intermediate value.

$$\begin{aligned} &\Theta(t) = f\left(\Theta_1, \Theta_2, t\right) \\ &\text{where} \\ &\Theta(t_1) = \Theta_1 \\ &\Theta(t_2) = \Theta_2 \end{aligned} \tag{3.1}$$

QuantumIon supports several modes of interpolation as shown in Figure 3.3. In this figure, a series of interpolation keypoints are defined, and each graph shows different resulting interpolated value as a function of sample number $n = \text{floor}(t/T_s)$.

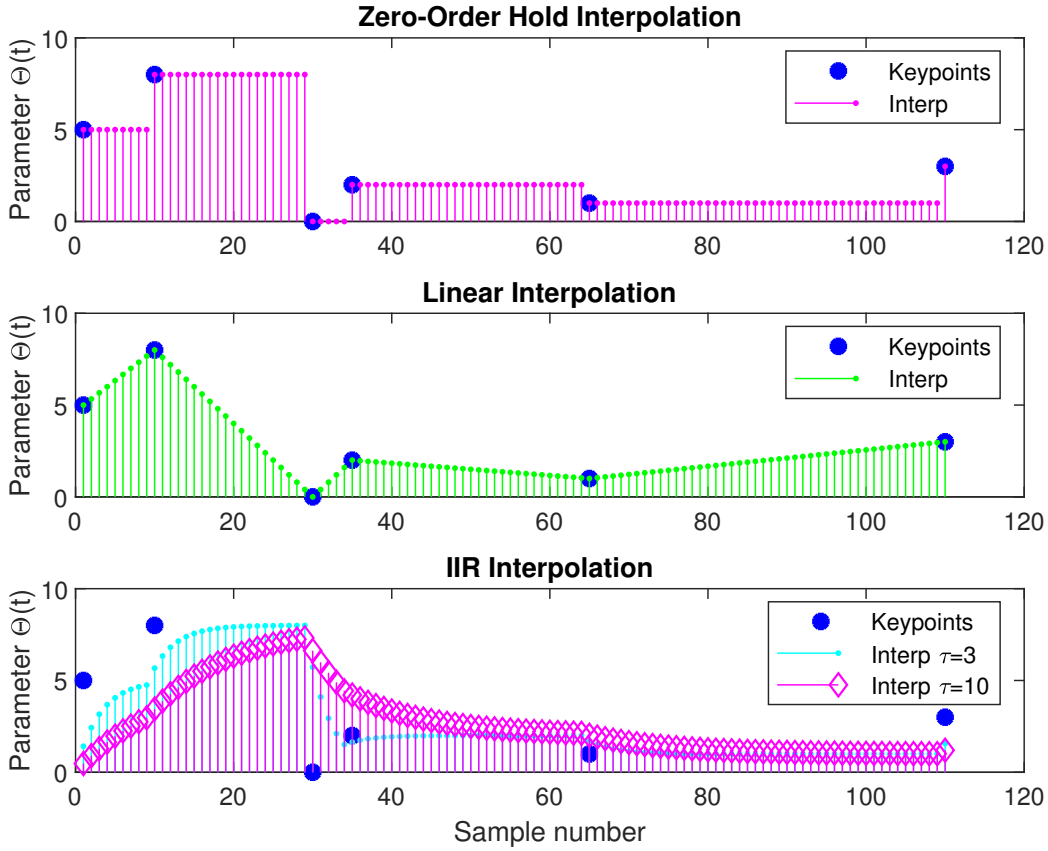


Figure 3.3: Interpolation Styles. Parameters in an FPGA, such as amplitude or frequency of an oscillator, can be programmed to vary smoothly over time. (Top) Zero-order hold polynomial interpolation holds the last value until a new value keypoint is given. (Middle) Linear interpolation smoothly varies in a straight-line trajectory between keypoints. (Bottom) IIR Interpolation simulates a bandwidth-limiting filter applied to zero-order hold. Note that IIR may miss the exact value of keypoints, but limits the effective sideband frequencies.

#### 3.2.5.1 Polynomial Interpolation

One possible interpolating function $f$ is the polynomial interpolation. Polynomial interpolation is computationally straightforward. For example, a simple piecewise-linear interpolator is generated as follows, using the relative time $\Delta t = t - t_1$ since the first interpolation

point:

$$\Theta(\Delta t) = \Theta_1 + \frac{\Theta_2 - \Theta_1}{t_2 - t_1}\Delta t. \tag{3.2}$$

The polynomial interpolation coefficients are given by the offset $a_0 = \Theta_1$ and slope $a_1 = (\Theta_2 - \Theta_1)/(t_2 - t_1)$. The piecewise-linear interpolator cannot guarantee smoothness at the interpolation points. QuantumIon improves performance by allowing third-order polynomials in relative time.

$$\Theta(\Delta t) = a_0 + a_1(\Delta t) + a_2(\Delta t)^2 + a_3(\Delta t)^3. \tag{3.3}$$

The third-order interpolation can provide smoothness (although not infinite smoothness) at the interpolation points. However, numerical instability can occur when the highest-order factor $(\Delta t)^3$ exceeds the 64-bit internal arithmetic of the FPGA. As a result, the maximum distance between interpolation points is approximately

$$\begin{aligned}(t_2 - t_1)_{max} &= \sqrt[3]{2^{64} - 1} \\ &\approx 2,640,000 \text{ samples} \\ &\approx 1.3 \text{ ms}.\end{aligned} \tag{3.4}$$

In contrast, the second-order term has a maximum time of approximately 2.1 seconds. This allows the user a trade-off between approximation error and the number of interpolation points by simply setting higher-order coefficients to zero. The polynomial interpolation is mathematically well-defined: The error bounds can be found by simply summing the remaining terms of a Taylor series expansion. Additionally, for some problems (such as a linear sweep of frequency), a polynomial evolution of parameters is directly tied to the problem at hand.

The first example of Figure 3.3 show a zero-order hold, where $a_0 = \Theta_i, a_1 = a_2 = a_3 = 0$. The zero-order hold corresponds to sudden snap of parameter changes. The second example of Figure 3.3 shows linear interpolation (i.e. first-order) between two points. Note that first and higher orders require the user to pre-program the appropriate coefficients, whereas zero-order hold does not, since the $a_{1,2,3}$ depend on the future keypoints.

#### 3.2.5.2 Bandwidth Limited Evolution

For some parameters, polynomial interpolation may not represent true physical processes that would disturb the Quantum system. Another possible interpolation is to constrain the rate of change of the parameter $\Theta$. This kind of parameter evolution is modelled as a linear time-invariant system system[32]. In a discrete-time system, such as those with Analog-to-Digital Converter (ADC) and DAC converters, the parameter evolves over fixed-time steps $t_n$ as

$$\Theta(t_n) = \sum_{j=0}^{M} a_j \Theta_{\text{in}}(t_{n-j}) - \sum_{k=1}^{N} b_k \Theta(t_{n-k}). \tag{3.5}$$

Equation 3.5 shows the next output parameter $\Theta$ is a weighted sum of previous outputs, and inputs $\Theta_{\text{in}}$. Such a recurrence relation forms a classic Infinite Impulse Response (IIR) filter. In this model, the input $\Theta_{\text{in}}$ is a zero-order hold between interpolation points $\Theta_i$. The IIR filter topology is very efficient since a relatively few number $M$,$N$ of coefficients can achieve a very smooth transition. QuantumIon limits the IIR order to second order,

$$\begin{aligned} \Theta(t_n) =& a_0\tilde{\Theta}(t_n) + a_1\tilde{\Theta}(t_{n-1}) + a_2\tilde{\Theta}(t_{n-2}) && (3.6)\\ & - b_1\Theta(t_{n-1}) + b_2\Theta(t_{n-2}) && (3.7) \end{aligned}$$

From a physical perspective, the coefficients determine the effective bandwidth of the parameter. For example, a first-order IIR can be written as

$$\begin{aligned} \Theta(t_n) &= a_0\tilde{\Theta}(t_n) - b_1\Theta(t_{n-1}) && (3.8)\\ &= (1-b_1)\tilde{\Theta}(t_n) - b_1\Theta(t_{n-1}), && (3.9) \end{aligned}$$

where the condition $a_0 + b_1 = 1$ ensures unity gain. This first-order recurrence relation is completely defined by the parameter $b_1$, and has an impulse response

$$\Theta(t_n) = \tilde{\Theta}(t_0)(b_1)^n. \tag{3.10}$$

Equation 3.10 shows a decaying exponential trend, with a time constant $\tau$ such that $(b_1)^\tau = e^{-1}$. Note that three such time constants effectively decays by 95% of the initial value. Such a simple first-order IIR operates similar to an electrical Resistor-Capacitor (RC) filter. For sampled data systems, the time constant analogy only goes so far, but the IIR system can still be configured to effectively limit the bandwidth of the parameter in question. For

more details on specifying coefficients for a given bandwidth, see Rorabaugh [33][4].

## 3.3 Execution Engine

Each function within an FPGA contains a dedicated execution engine. The purpose of the execution engine is to perform the operations of the execution flowgraph (see Section 4.4). The execution engine supports several concepts from traditional computing machines:

- A *program counter* which contains the current node in the execution graph,
- A series of *operation codes* (opcodes), which define what operations should take place at the epoch,
- A single *loop counter* which contains a count-down variable to enable loops,
- An *execution epoch table* which defines when in the experiment a particular opcode should take place,
- A *branch lookup table* (LUT) which forces changes to the execution graph based on measurements from the quantum machine.

Each major function is described in the following sections.

### 3.3.1 Looping

The looping construct is important to understand since it takes up a large number of opcodes to implement. Consider the following code fragment:

```
int n=100;
while(n > 0)
{
  do_many_things();
  n = n-1;
}
do_end_stuff();
```

Listing 3.1: Pseudocode for a loop

[4] A careful reader may notice that the IIR response does not necessarily hit the interpolation keypoints as the polynomial coefficients may. Guaranteed keypoints are not a feature of time-invariant systems.

Under compilation to low-level operation codes[5], the following assembly code instructions are generated:

```
start:
    set R0, 100;
loop_top:
    do_many_things;
    add R0, -1;
    test R0;
    jump_if_nonzero loop_top;

    do_end_stuff;
end:
```

Listing 3.2: Opcodes for a loop

The looping operation requires several basic constructs: the ability to set a counter value, the ability to test the value of the loop counter, the ability to jump based on a condition, and the ability to decrement the counter. The example in Listing 3.1 and Listing 3.2 is somewhat simplistic, but most cases can be covered with a few more opcodes. The full list is described in Subsection 3.3.4.

### 3.3.2 Program Counter

The program counter is an index to the table of instructions. It represents the current node in the execution graph such as those of Figure 4.5. Normally, the program counter increments at the end of each instruction; the *jump* opcodes alter the program counter. The program counter is global; each FPGA in the entire QuantumIon system is running at the same program counter value, with identical copies running in synchronization on each FPGA card.

### 3.3.3 Execution Epoch Table

The execution epoch table corresponds to physical times in which opcodes are executed. Unlike the program counter, which is an index representing *which* instruction to execute,

[5]Opcodes and assembly language instructions are generally synonyms, although opcodes are binary encodings while instructions are the human-readable equivalent names.

the execution epoch table contains physical *times* in half-nanosecond increment. At each execution epoch, the opcode is executed and the program counter is incremented. The execution epoch is global; the same epoch table is shared in every FPGA in the QuantumIon system. In this way, the execution epoch is decoupled from the action that are performed.

### 3.3.4 Operation Codes

Operation Codes (Opcodes) represent the most basic programmability of the QuantumIon system. Each module's engine must respond to a set of opcodes as listed in Table 3.1.

| **Name** | **Function** |
|---|---|
| `SetValue` $v$ | Sets the value of a programmable function to $v$ |
| `SetLoop` $v$ | Sets the loop counter to $v$ |
| `JumpLoopZero` $l$ | Moves to a new location $l$ in the epoch and opcode tables if the loop count is zero |
| `JumpLoopNonZero` $l$ | Moves to a new location $l$ in the epoch and opcode tables |
| `DecLoop` | Decrement the loop counter |
| `BranchLookupTable` $t$ | Jump to a location based on the lookup table $t$. |
| `Goto` $l$ | Jump to a location $l$ |
| `NoOp` | No operation (does nothing) |

Table 3.1: Execution Opcodes. At the FPGA execution engine, each time instant corresponds to only one of these operations. The `SetValue` opcode indicates the change to a parameter, such as the on/off state of a TTL output. There are a number of opcodes to support looping constructs. The `BranchLookupTable` opcode supports the branching logic for partial measurements. Finally, the `NoOp` opcode is used to provide no action; such no-action is often used when the time instant corresponds to an action on another card.

The `SetValue` opcode is the primary workhorse of the execution engine. When the program counter matches this instruction, a new function value is engaged. Examples of new values may be the setting of a TTL output voltage (on or off), a new feedback controller setpoint, or magnetic field current.

The `SetLoop` opcode sets the loop counter to a fixed value. This is generally performed at the top of a loop.

The `JumpLoopZero` opcode jumps to a new program counter location only if the loop counter is zero. This is generally performed at the bottom of a loop.

The `JumpLoopNonZero` opcode is the logical opposite of the `JumpLoopZero` opcode. It is the alternative used at the bottom of a loop.

The `DecLoop` opcode decreases the value of the loop counter by one. It is generally used at the bottom of the loop.

The `BranchLookupTable` opcode also changes the program counter, but it is based off a classical-quantum measurement. The user specifies the particular lookup table, and this table dictates the next program counter value based on the measurement. See Subsection 3.3.6.

The `Goto` opcode performs an unconditional change of the program counter. It is generally used at the bottom of a loop to return to the top of the loop.

The `NoOp` opcode performs no operation. `NoOp` is the most common opcode in a typical quantum program. Recall that the execution epoch table, program counter, loop counter and so on are global. However, in a typical execution flowgraph such as Figure 4.6, each node represents only a few actual actions. Because of the global nature of the flowgraph, most modules perform no action at a node. The `NoOp` instruction ensures such no-action is possible. An example is given in Table 4.1.

### 3.3.5 Loop Counter

The loop counter is a variable in memory that contains the value of the current iteration of the loop. The loop counter is global throughout the QuantumIon system, and every FPGA module within quantum program shares the same conceptual value. In reality, several copies of the loop counter are distributed around each FPGA so that each may run independently. The consistency of the execution flowgraph ensures that each FPGA is running identically despite the multiple copies of the loop counter.

### 3.3.6 Branch Lookup Table

The branch lookup table is used to perform branches in the execution flowgraph. Each measurement-capable module (such as an image processor of Subsection 3.4.7, or PMT counter in Subsection 3.4.2) shares an *measurement result* in the form of a classical word. For binary systems, this word is a one-to-one mapping. For nonbinary quantum programs such as three-level *qutrit* systems, a suitable encoding has still to be defined.

An example of such a nonbinary encoding could be the binary value equivalent to the nonbinary digit value. For a $d$-level system with partial measurements $m_n$, the value of

the encoded word $W$ could be:

$$W = \sum_n m_n d^n. \tag{3.11}$$

The value $W$ in Equation 3.11 can be encoded as a standard binary value and used in an FPGA lookup table.

The measurement word $W$ is broadcast to all FPGA modules by the measurement module. In early versions of the QuantumIon hardware, this broadcast is over the PCI express bus (see Subsection 3.2.3). In larger versions of the system, the broadcast is to be over Infiniband (see Subsection 3.2.4).

The measurement word is received by all FPGA modules and shared by the execution engines of all modules. When the branch is taken (i.e. when the epoch corresponding to a `BranchLookup` opcode is reached), the measurement word is used as an index to a lookup table, and the corresponding value is taken to be the new program counter value[6].

## 3.4 FPGA Modules

Each major function is comprised of an FPGA Module. *Module* is a generic term for an engine of precise-time logic. Several copies of the same FPGA module may be present on one FPGA card, and the same card may have multiple different modules. An FPGA module is similar to a 'function' in typical computer programming.

### 3.4.1 Discrete (TTL) Output Module

The discrete TTL output is shown in Figure 3.4. The primary purpose of the TTL output user block is to generate precisely-timed on-off voltages to drive switches, actuators and the like. The TTL output block is programmed with a series of `Value` fields at precise times. Each `Value` is either logical one, or logical zero. These correspond to high voltage (e.g. +5VDC) or low voltage (e.g. Ground). When the experiment clock value reaches the an opcode corresponding to a value change, that output is changed immediately. Each physical output has its own execution engine. In this way, although the FPGA code may be the simplest of all in this particular module, the FPGA resource demand is high.

[6]Note that there is a distinct separation between the measurement epoch and that when the branch is taken. This is to account for the latency in the measurement as well as the broadcast of the new word.

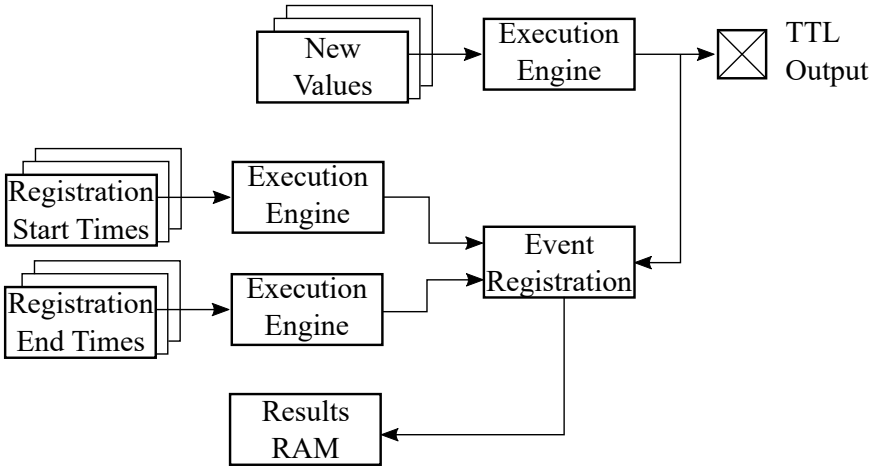


Figure 3.4: TTL Output Module Concept. The TTL output is a simple on/off output, with a recorder to diagnose when transitions actually occur. The value of the output, and the start and end times of the event recorder are attached to execution engines for precise time controls.

### 3.4.2 Discrete (TTL) Input Module

The discrete TTL input module is shown in Figure 3.5. TTL inputs are used to provide two main functions: pulse counting (such as from PMTs), and timing registration of discrete events.

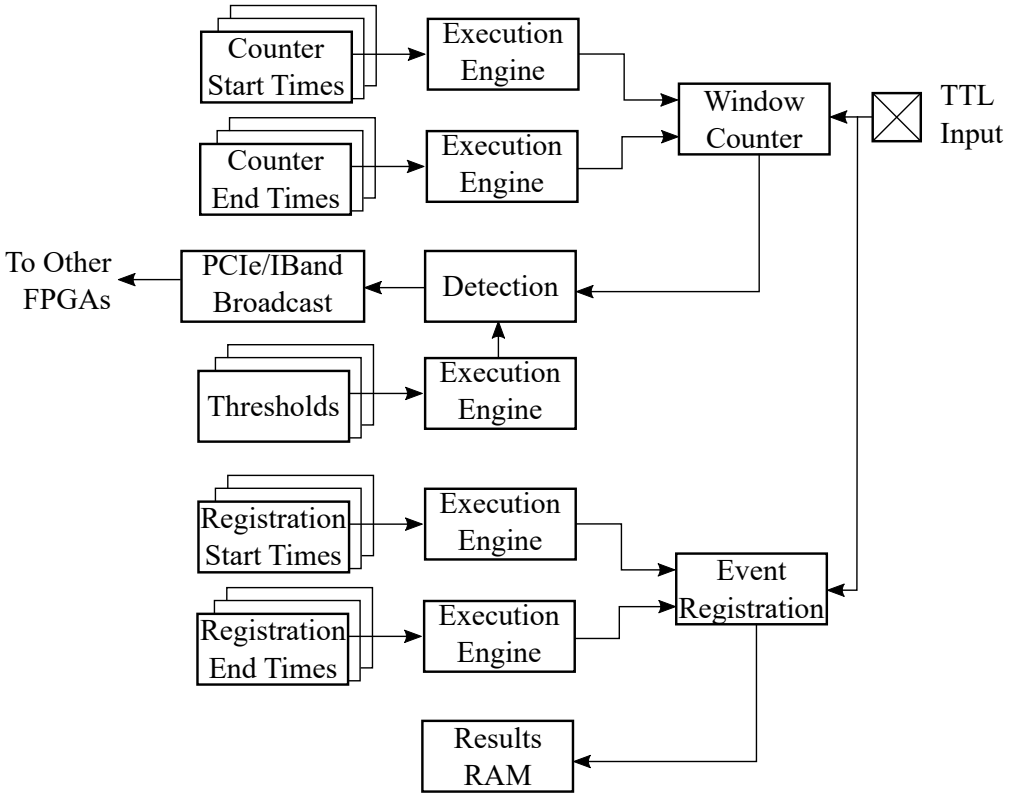


Figure 3.5: TTL Input Module Concept. Each TTL input is connected to a window counter with programmable start and stop times. The input transitions are recorded by the event registration engine to give debugging similar to an oscilloscope. The window counter is also connected to a detection module with programmable threshold to be broadcast to other FPGAs for branching logic.

In pulse counting mode, the TTL input engine is given a "window" starting with the precise time and duration. During this time, the counter increments its state by one for each voltage transition. At the end of a count window the result is saved as an identified resource for later download by the user.

In timing registration mode, each transition is recorded along with the experiment clock time it occurred. In this way the user may calculate time-of-arrival and phase information for different measurements. The user provides a maximum number of events to be counted as part of the resource allocation (see Section 6.4).

Each mode has separate start and end times to enable the mode.

### 3.4.3 Analog PID Module

The analog Proportional-Integral-Derivative (PID) controller module is the most basic linear feedback controller. Such feedback controllers are described in detail in Chapter 7.

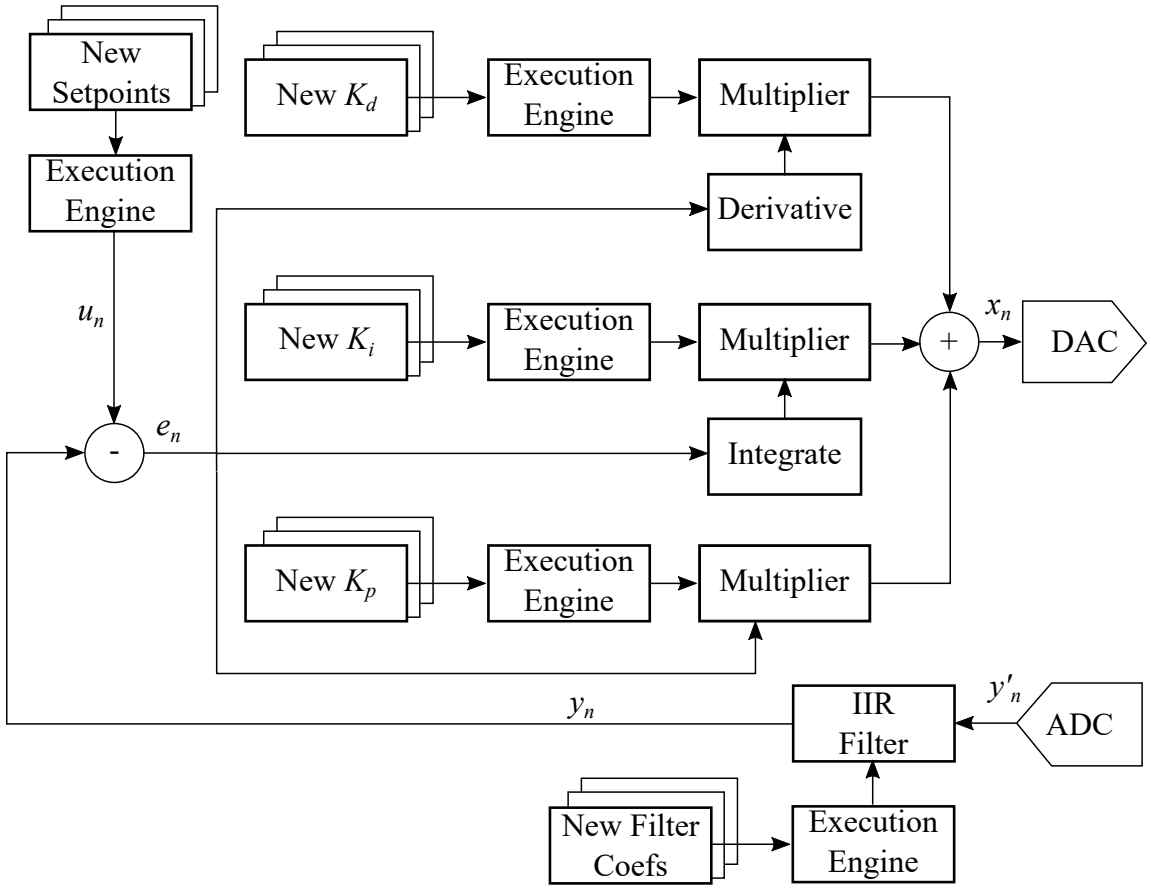


Figure 3.6: Analog PID Module. The basic feedback engine measures the difference between a programmable setpoint and a feedback signal. The proportional, integral, and derivatives of this error, with programmable coefficients become the new output. The output is converted to an analog voltage by a Digital-to-Analog converter. Feedback from a sensor is received by the Analog-to-Digital converter, and filtered prior to being used to compute the error signal. The filter coefficients are also programmable.

The FPGA implementation of the PID controller is shown in Figure 3.6. The *setpoint* signal is the primary adjustment in this module, whereas the coefficients $k_i$, $k_p$, and $k_d$,

while adjustable in the same way, are likely to remain static over the course of a quantum program. The various controls change under the control of execution engines as described in Section 3.3. The *output* signal is converted by the DAC into a voltage for use in the QuantumIon electronics, while the *readback* signal is the digitized sensor voltage read by the ADC. The controller implements a second-order IIR filter for noise cleanup[7]. For a setpoint $u$ and readback signal $y$, the module implements the feedback compensator $G(s)$, which gives an output $x$

$$G(s) = \left(k_p + k_d s + \frac{k_i}{s}\right) e(s), \tag{3.12}$$

where $e(s) = u(s) - y(s)$ is the Laplace transform of the error signal. Since this controller is implemented in a sampled-data system, the Laplace transform $\mathcal{L}$ becomes the $z$-transform [32]

$$\mathcal{L}(y) = \int_0^\infty y(t)e^{-st}dt \quad \rightarrow \quad \mathcal{Z}(y) = \sum_{n=0}^{\infty} y_n z^{-n}. \tag{3.13}$$

The three terms of Equation 3.12 correspond to the *proportional* term $k_p e(s)$, *integral* term $(k_i/s)e(s)$, and *derivative* term $k_d s e(s)$ are implemented digitally as the transfer function $G(z)$

$$G(z) = \left(k_p + z k_d + z^{-1} k_i\right) e(z). \tag{3.14}$$

For an error signal calculated as $e_n = u_n - y_n$ corresponding time-domain output $x_n$ is

$$x_n = k_p e_n + k_i \left[e_n + e_{n-1}\right] T_s + k_d \left[e_n - e_{n-1}\right] / T_s. \tag{3.15}$$

Finally, the IIR filter is used to remove noise and interferers from the readback signal. It implemented as a simple recursive function,

$$H_{filter}(z) = \frac{y(z)}{y'(z)} = \frac{q_2 z^{-2} + q_1 z^{-1} + q_0}{p_2 z^{-2} + p_1 z^{-1}}, \tag{3.16}$$

where the coefficients $p_k$ and $q_l$ create the poles and zeros of the transfer function (for a full analysis, see [32] and [33]). Note that coefficients are adjustable over the course of a quantum program, but are unlikely to change in practice. The corresponding time-domain

[7]The IIR filter is not a critical part of the analysis, since it is intended for use only to remove interference. Thus the readback is considered to begin *after* the filter. Provided the implemented filter is approximately low-pass, and has modest phase delay this assumption is reasonable.

implementation is given by

$$y_n = q_2 y'_{n-2} + q_1 y'_{n-1} + q_0 y'_n - p_2 y_{n-2} - p_1 y_{n-1}. \tag{3.17}$$

### 3.4.4 Direct Digital Synthesis (DDS) Module

The DDS module performs synthesis of the RF signals to drive AOM components, which in turn modulate, stabilize and tune each laser beam. Unlike the AWG module described in Chapter 5, the DDS module provides a pure sinusoid at precise frequency, phase and amplitude. Such synthesis is well suited for use in stabilization loops and other non-user-programmable controls.

The DDS concept is a simplification of the CORDIC engine described in [34][35]. The core of the DDS is the so-called *Givens Rotation*. In this rotation, a state $[x, y]^T$ of unit length may be rotated as

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \\ \sin\theta_i & \cos\theta_i \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}. \tag{3.18}$$

The state $[x, y]^T$ can be considered the real and imaginary components of a complex exponential $e^{i\theta}$. Synthesis of a frequency $f$ at sampling rate $F_s$ is then just a repeated application of the Givens matrix by angle $\theta_i = 2\pi f/F_s$. The output is simply the real component $x$ of the state. The quantity $\tilde{f} = f/F_s$ is sometimes called the *digital frequency*.

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} \cos 2\pi\tilde{f} & -\sin 2\pi\tilde{f} \\ \sin 2\pi\tilde{f} & \cos 2\pi\tilde{f} \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}. \tag{3.19}$$

The elements of the Givens matrix for a given frequency are trigonometric functions. However, for a given frequency these elements are constant. This is attractive for FPGA implementation since the trigonometric calculation can be performed in the main server, using a full suite of mathematical libraries. Precision-timed frequency modulation can thus be performed by changing the value of the Givens matrix over the course of the experiment.

Phase jumps can be set in a similar way. The state $[x, y]^T$ is the instantaneous complex exponential. Therefore a jump to a particular phase $\phi$ is accomplished by forcing the state to a particular value

$$\begin{bmatrix} x_\phi \\ y_\phi \end{bmatrix} = \begin{bmatrix} \cos\phi \\ \sin\phi \end{bmatrix}. \tag{3.20}$$

During a quantum experiment, the ability to alter the phase is given by a list of new phase values and the corresponding times of execution. At the prescribe instant, the running phase $[x_i, y_i]^T$ is switched out, and replaced by a new absolute phase $[x_\phi, y_\phi]^T$. This one-time event forces a phase change, after which the system returns to the running phase. A continuous-wave sinusoid with discrete phase jumps can be easily programmed in this manner.

The state $[x, y]^T$ is a normalized unit vector. To perform amplitude modulation, the real output $x$ is passed through a standard hardware multiplier. The coefficient of this multiplier is a list of the real amplitude of the system. Similar to the phase and frequency controls, an amplitude change is given by a new value for the multiplier at a precise time. Unlike the phase control, the amplitude control is held constant until a new value is programmed.

The conceptual block diagram is shown in Figure 3.7. For each control (amplitude, frequency and phase), a separate execution engine feeds changes at precision times. Each control is associated with a memory array of new values. For the phase control, the array contains a list of absolute state variable $[x_\phi, y_\phi]^T$ that are engaged by the one-time switch. Similarly the 2x2 matrix multiplier is associated with an array of Givens matrix coefficients. Finally the scalar multiplication stage extracts the real state variable $x_i$ and multiplies it by the amplitude value in the associated amplitude array. The output of the scalar multiplier is sent to the DAC for output to the associated amplifier and AOM.

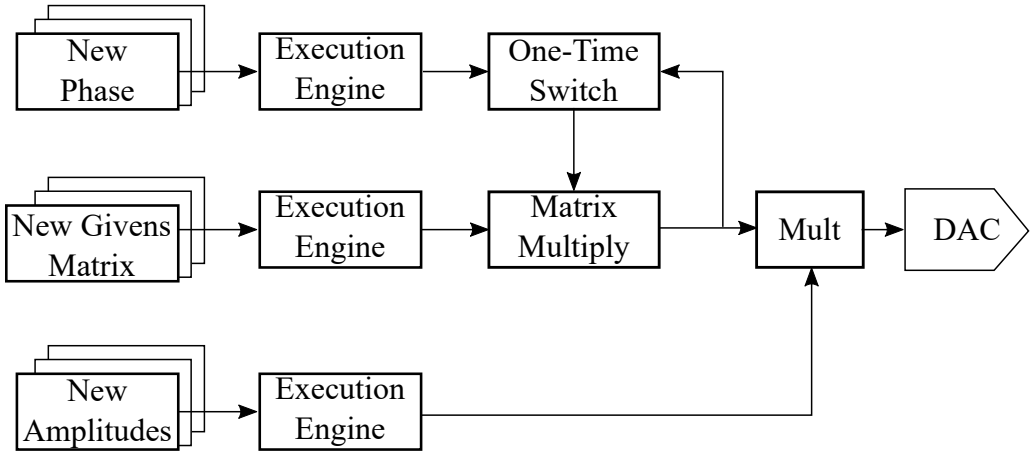


Figure 3.7: DDS Module Concept. The output sine wave is generated by recursive application of a Givens matrix. The instantaneous phase is a 2-element vector corresponding to the real and imaginary parts of a unit vector. Amplitude control is by a coefficient of the real part of the vector. Phase jumps and bumps are applied by changing the phase state variable, or by a one-time matrix operation.

### 3.4.5 Arbitrary Waveform Generation Module

The primary purpose of the AWG module is to provide pulse-shaping for the Raman laser beam. The AWG module is described in detail in Chapter 5.

### 3.4.6 Amplitude Stabilization Module

The RF Amplitude stabilization module ensures a constant radio-frequency power is delivered to associated devices such as the trap RF resonator module. Amplitude stabilization requires the periodic adjustment of the amplitude of the generated RF signal, and as such a complete, stabilized RF output is connected directly to the DDS module that generates such a signal.

The primary components of amplitude stabilization are shown in Figure 3.8. The target amplitude setpoint is input to a PID compensator, which controls the DDS core. Details of the DDS core are given in Subsection 3.4.4. The DAC module creates the RF waveform's instantaneous voltage, which is fed to a power amplifier. Such a power amplifier is assumed

to a frequency-dependent gain[8]. The amplifier's high-power output is sampled with a directional coupler which provides an RF signal back to the FPGA via the ADC module. Since the sampled signal is an RF carrier (or has some other modulation), the received signal must be demodulated, and the Root-Mean-Square (RMS) power is estimated. The difference between the received power estimate and the desired setpoint are input to the PID compensator, thus closing the feedback loop.

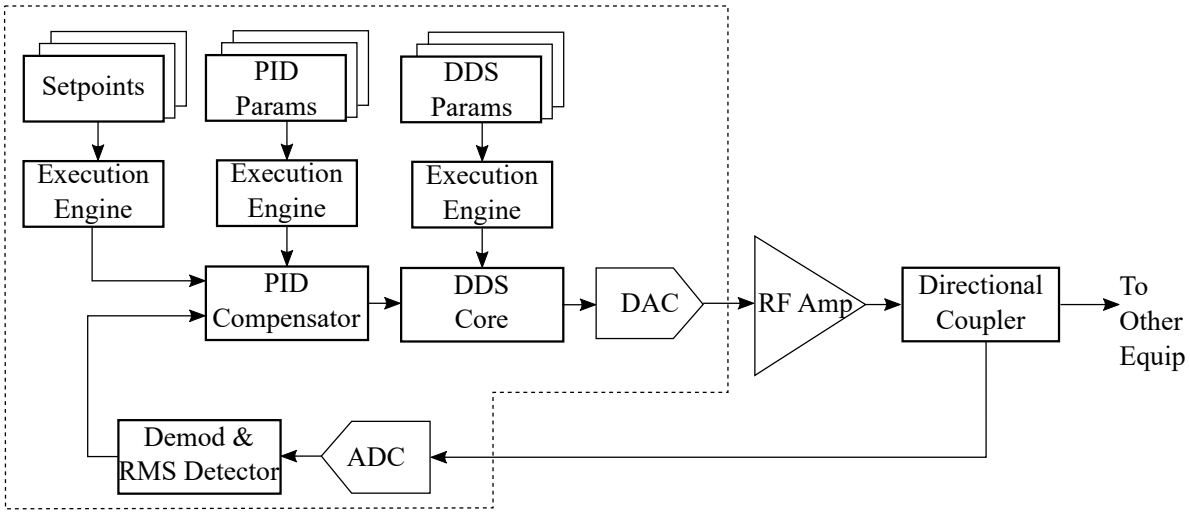


Figure 3.8: RF Amplitude Stabilization Core. Amplitude stabilization consists of a PID controller and DDS core for signal generation. The amplified RF signal is detected by a directional coupler for feedback. The output is stabilized against variations in amplifier power and losses.

### 3.4.7 Image Processing Module

The image processing module receives all data from the CCD cameras around the Quantum-Ion equipment. Each camera is connected via a dedicated internal Ethernet network using the GigE specification, or, in the case of the primary imaging camera, via the CameraLink HS protocol. Image processing modules perform two basic types of operations, as shown in Figure 3.9.

[8]Other fluctuations, such as drift with temperature, or ageing, are compensated by this module.

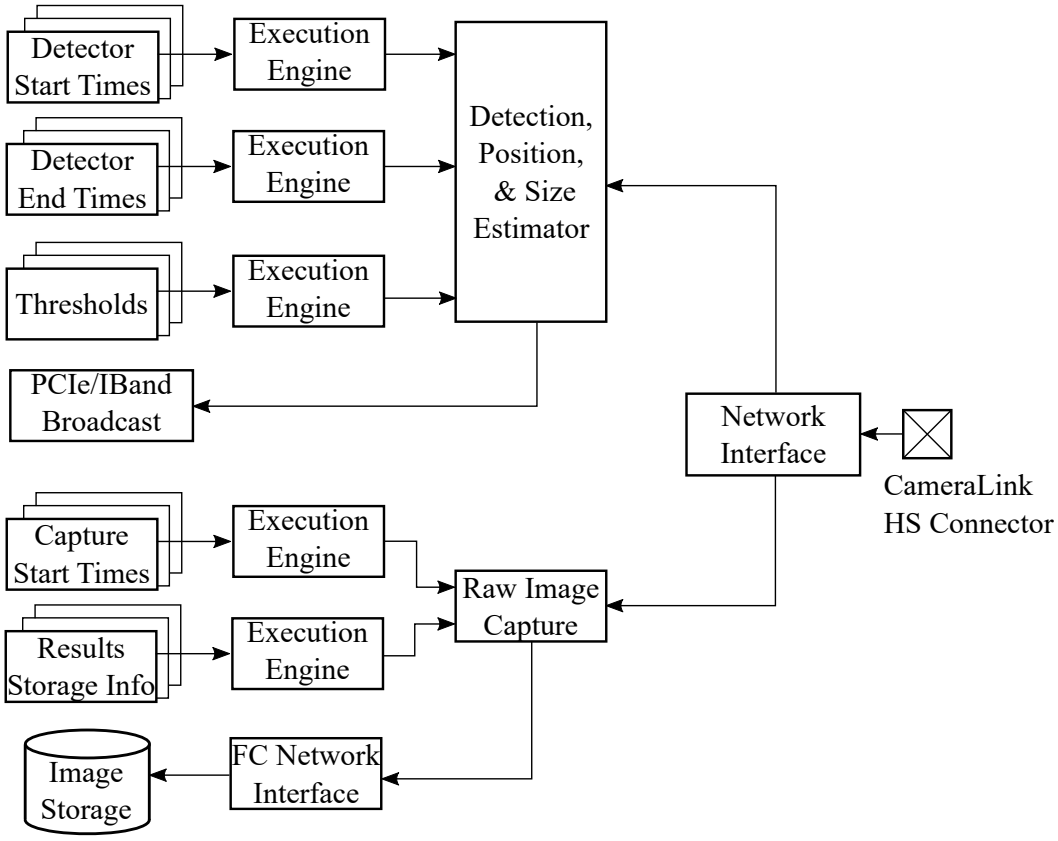


Figure 3.9: Image Processing Core. The camera image is received over the CameraLink HS interface. Two processing engines provide for the capture of raw images, and for real-time detection of ions. The realtime estimator has programmable start and end times, and programmable threshold. The results are broadcast to other FPGAs via PCI Express or Infiniband. Raw image capture has programmable capture start time, and a temporary results RAM, as well as direct connection to a storage drive array via Fibre Channel.

Firstly, the image processing module stores raw images onto a dedicated Fibre Channel (FC) network. The resource concept described in Section 6.4 identifies each image for later download. Secondly, the image processing module, in conjunction with the TTL input counter described in Subsection 3.4.2, performs quantum state detection. To this end, a series of start/stop times and corresponding detection thresholds are executed at precision times. The result of the detection decision is broadcast via PCI express (Subsection 3.2.3) or Infiniband (Subsection 3.2.4) to all other cards for use later.

### 3.4.8 Shuttling DAC Module

The ion trap provided by Sandia National Laboratories is a planar surface trap [13]. This trap contains approximately ninety electrodes requiring DC voltages to provide the static potential for ion confinement. To provide this, a custom DAC module provides individual analog outputs to each electrode as shown in Figure 3.10. Since each DAC is independently controlled, physical ion motion such as *shuttling* can be performed by sequentially changing electrode voltages.

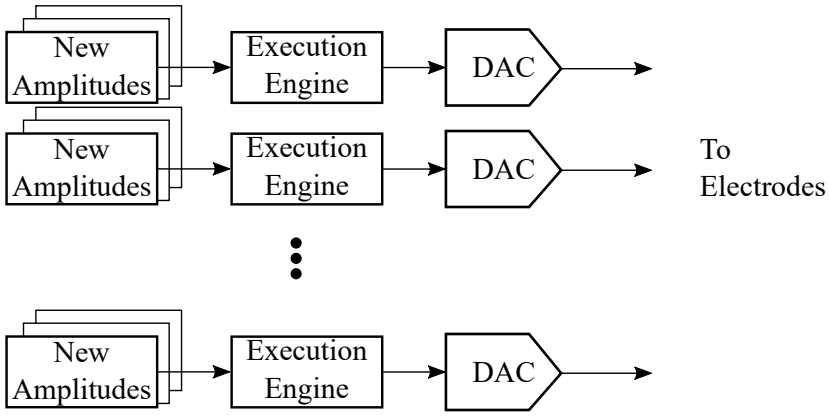


Figure 3.10: Shuttling Module Core. The shuttling module is a collection of up to 50 Digital-to-Analog converter modules per FPGA. Each output is connected to an electrode in the ion trap. Each DAC is given a unique programmable amplitude, with precision timing provided by an execution engine.

In some respects, the Shuttling DAC is conceptually simple: each channel has an independent execution engine that feeds new voltage amplitudes to the DAC. However, the large number of channels requires a high-performance FPGA despite the simple function.

### 3.4.9 Conclusion

This chapter described the technical internals of the programmable FPGA hardware, and its associated programming. Each module shares a common interconnect scheme, execution engine, and sample clock. FPGA modules provide a series of unique functions, one for each physical card; these functions are also described in detail. The collection of these unique functions, along with sampling, communication and triggering, provide the lowest-level hardware control of QuantumIon.

Note that the direct programming of these FPGA modules is *not* accessible to the user; such control would be dangerous for all but the QuantumIon core team. Instead, QuantumIon's power comes from the use of execution engines for each functional parameter. Controlling the parameters *is* under the users' control. This separation allows the user to focus on the timing aspects of a quantum program, instead of how the hardware is to work together. It also allows the QuantumIon system to provide safety checks, overrides, and other abilities to provide a suitable platform for shared use.

The FPGAs' execution engines are programmed by the other major software of QuantumIon: the main program residing on the system server that will be described in the next chapter. Unlike FPGA programming, the main program is designed in a way familiar to most computer programmers. The main program provides the first interaction with the end user, and as such requires high-level language, a full operating system, network connectivity, security and a host of other functions.

# Chapter 4

# Main Control Program

Previously, the lowest level of programming and control was described: that of the FPGA modules. These modules handle all precision timing operations for QuantumIon. In this chapter, we describe the high-level programming environment: the programming of the main Linux server. This program performs many different operations, most importantly that involving interacting with the end user, and the compilation of quantum programs. Quantum programs use the XML language described in Chapter 6, and so the main program must translate these into the FPGA opcodes described previously.

The main control program performs most high-level operations in QuantumIon. In contrast to the FPGA hardware mentioned in Chapter 3, the main program has the full feature set of a Linux-based server. The program can thus use standard programming data structures, dynamic memory, advanced languages such as C++, and libraries for interfacing databases, high-speed I/O, and cryptography.

The main program is comprised of several major components discussed below. They are: the security layer, the program scheduler, the sequence compiler, the execution engine, the calibration database, data transport, and the symbolic algebra system.

## 4.1 Security Layer

As a publicly-accessible device on the Internet, the QuantumIon platform is susceptible to attacks. As a result, security must be considered in the early stages of design. Best-practice security is more than simple encryption (e.g. Secure Shell (SSH)). Instead, an entire security posture should be developed, and must address the following needs:

- Access Control: Each control, such as database write access, or control of a damaging laser, is only accessible to a limited number of users. All controls have a customizable Access Control List (ACL).
- Authentication: Only users whose identity has been verified in a secure way can have any access.
- Privacy: Communications between the user and the QuantumIon system is safe from eavesdropping.
- Transient credentials: Users' credentials, used for login and authentication, must be periodically renewed, and may be revoked without the users' participation. This prevents the sharing of credentials (e.g. sharing a file on the Internet), and allows for temporary elevation of privileges.
- Principle of least privilege: Users are granted the smallest set of privileges to get their work done. This prevents granting potentially dangerous permissions to most users, and limits the possibility of exploitable loopholes through compromised accounts.
- Role-Based Access Control (RBAC): Permissions are grouped into "roles", and users are members of a role. This contrasts with assigning elevated permission to individual users. ACLs are implemented with roles.

### 4.1.1 Transport Layer Security

QuantumIon uses modern Transport-Layer Security (TLS) [36] for session security. TLS is familiar to most readers as the `https://` website address moniker used in banking and e-commerce. The TLS protocol provides authentication, privacy, and transient credentials. In the TLS protocol, X.509 Certificate files are exchanged between the user's computer and the server. This certificate file used to login and transfer data to QuantumIon.

Certificates are public, and generally only one certificate is associated with a user (or server). Certificate files contain a series of cryptographic signature chains. A user (or server) first creates a cryptographic key pair using a common public key cryptosystem such as RSA. The key pair contains a private key (never shared), and a public key, which is known to all. The certificate contains a copy of the public key, and so a certificate can be used to decrypt data and validate the source of data (sender's identity), but it cannot be used to encrypt. Since the user has the only copy of the private key, only the *true* user

can encrypt new data. Thus, certificates can be used to provide an organized security layer known as Public Key Infrastructure (PKI) [1].

The certificate use process is shown in Figure 4.1. To create a new certificate, the user provides identifying information, a copy of its public key, and signs a Certificate Signing Request (CSR). This digital file is sent to a Certificate Authority (CA), who approves and creates a signed certificate file and expiration date. The user encrypts its identity using the private key, and sends this information along with a copy of the certificate to QuantumIon during login. QuantumIon checks the certificate's CA signature, and looks at a database of revoked certificates. Lastly QuantumIon attempts to decrypt the identity block using the certificate. If these checks succeed, the user is considered authentic. Since public key algorithms such as RSA are somewhat slow, after authentication a new cryptography scheme is negotiated for speed.

The concept of *trust* is important to understand. The server (QuantumIon) trusts a series of well-known CAs. Because of this trust, if a certificate is signed by a trusted CA, hasn't been revoked, and decrypts the identity block, then QuantumIon can *trust* the user.

[1] PKI is not to be confused with public-key cryptography standards such as Rivest-Shamir-Adleman (RSA). PKI refers to the trust chain and certificate authority *infrastructure* as a whole. PKI is not an encryption algorithm.

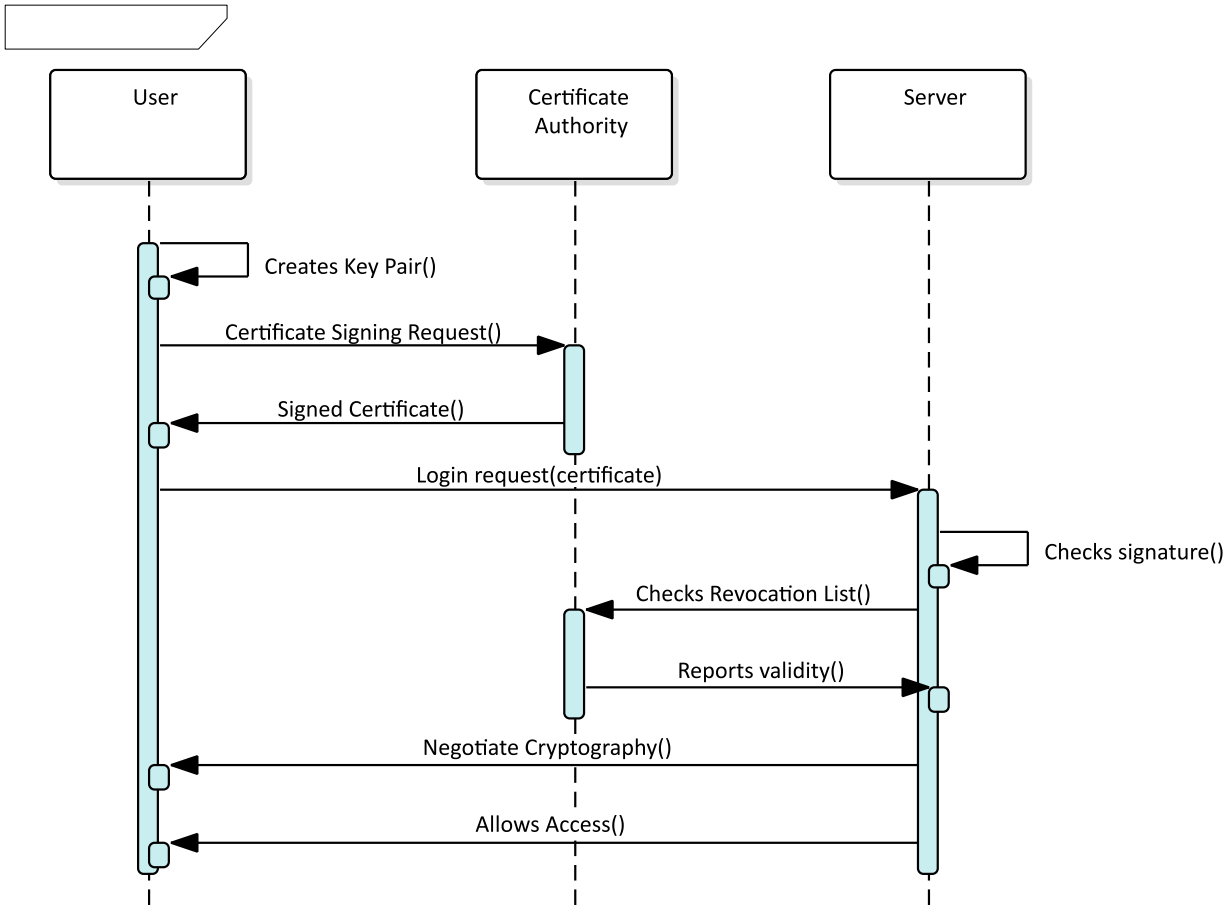


Figure 4.1: Simplified Public Key Infrastructure (PKI) protocol. Three entities are involved: a user, a certificate authority, and the consuming server. The user is responsible for creation of keys. The certificate authority signs the public key, creating a certificate file. The server, when it receives a login request from a user, checks the signature of the certificate with the certificate authority. If successful, the user is allowed access. The server must implicitly trust the certificate authority, but should not trust login requests until verified in this manner.

### 4.1.2 XML Interceptor & Application Server

QuantumIon is designed to be connected directly to the Internet, and best security practices assume the Internet to be a hostile environment. It it thus advisable to isolate QuantumIon from the whole Internet. Additionally, a typical user is likely to be accessing the

QuantumIon environment from a campus or corporate environment; such enterprise-grade IT environments often severely limit the types of Internet services that can be accessed – not every protocol is allowed.

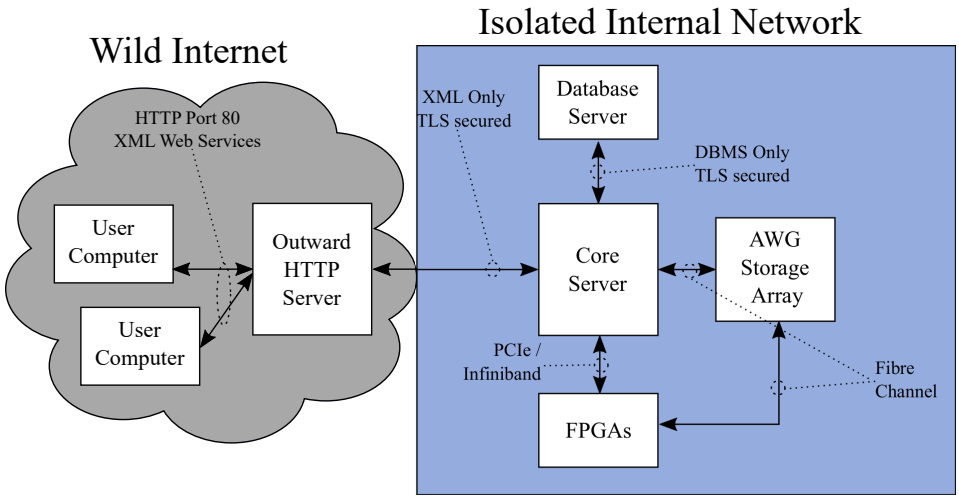


Figure 4.2: Server Network Architecture. The internal network is isolated from the Internet by a HTTP server, which operates as a firewall. The HTTP server only passes properly-formatted XML in the QuantumIon language, and the core server, which runs the main program, does not connect to anyone but the HTTP server.

As a result of these two security concerns (assumed hostility, and connectivity limitations), the QuantumIon main server is actually three different devices[2] as shown in Figure 4.2. The outermost server is the only one that connects to the Internet directly, and is what the user talks to when accessing QuantumIon. This server contains two separate network interfaces, and translates messages to the internal server which performs all further features; in this sense, the outward-facing server is merely a buffer. The final server, the database server, contains all calibration, saved user resources, and other long-term storage information with the exception of AWG waveforms as described in Chapter 5.

The outward facing server performs an additional function that addresses the issue of limited user connectivity. The most common protocol allowed on nearly every IT environment, from government to academia, is the familiar Hypertext Transport Protocol (HTTP) protocol used by web browsers. This protocol is most often associated with the transfer of

[2]This may be three physical servers, or virtual instances on the same physical machine. The two are equivalent.

web pages, however it is also used commonly and robustly to transfer raw XML commands for remote control and download of data, through the use of so-called *web services*[37]. Web services are supported by a variety of binding languages, including Python, Matlab, and Mathematica. The web service simply provides a standard way for the user to connect using XML, and accessed using the HTTP protocol. Because HTTP is often the target of malicious actors, the outward facing server (which is, in fact, a HTTP server) is suitably isolated from the QuantumIon core server.

HTTP is an excellent choice for the connectivity to the user, however the protocol itself is somewhat complicated to implement fully and robustly. Yet another use for the outward facing server is to perform the role of XML translation, whereby the received XML from a user is stripped of the associated HTTP information and streamed directly to the core server using a very limited channel. This is advantageous in that the simple protocol between the core server and the outward-facing HTTP server can be robustly secured, and the core server can only accept connections from a specific server; such a scheme is a standard way of hardening a server environment. Additionally, the core server need only implement this simple XML transfer protocol, and not burden itself with a more complicated protocol like HTTP.

## 4.2 Sequence Compiler

The sequence compiler converts the intermediate XML user language described in Chapter 6 into the execution engine language described in Section 3.3. The sequence compiler should be distinguished from quantum gate compilers, which generate quantum operations from a higher-level language describing a quantum algorithm. In QuantumIon, the sequence compiler has no knowledge of quantum operations at all.

The sequence compiler must perform many sub-tasks to transform the XML user program into the FPGA opcodes. These are performed roughly in the following order:

- Decrypt and expand any encrypted third-party programs as described in Section 6.7.
- Expand the sub-functions described in Section 6.6.
- Retrieve the current calibration database and solve symbolic algebra expressions.
- Calculate the total expected run time, and validate it is within the maximum permissible range.

- Convert absolute timing information to relative time.
- Allocate concrete storage for stored resources such as counters and images.
- Build lookup tables for branching logic.
- Generate the opcode sequences for each FPGA module.
- Validate the user has permission to change the controls as requested (validation of action tags).

### 4.2.1 Decryption of Third-Party Programs

Decryption of third-party programs is described in Figure 6.1 of Section 6.7. In this phase, the sequence compiler retrieves the private key specified by the `<keyHash>` tag within the third party library block. The encrypted block contained within the `<cipherText>` tag is expanded[3] to its cleartext equivalent. The result of this phase is a new XML program, or sub-function, which is as if it were programmed in the clear. The encrypted and unencrypted programs are shown in Listing 4.1 and Listing 4.2.

[3] QuantumIon uses the open-source `openssl` libraries, as such the cipher types are limited to that suite.

```
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  <resources>
    <pmt-counter>
      <id>counter1</id>
      <uuid>df271d2d-5c44-4f3d-9efc-7529839e8dbe</uuid>
    </pmt-counter>
  </resources>

  <headers>
    <function-header name="thirdparty-func-1"/>
  </headers>

  <functions>
    <secure-library uuid="b2c3f0af-489a-4069-9581-4565ac8ba14d">
      <keyHash type="sha256">a8add4003bf9e1eed2aca713db9bacc0d453625f001e3d5353ac748df920c994</keyHash>
      <cipherText cipher="aes-256-cbc">
        <![CDATA[
          U2FsdGVkX1+D7jkpaqWzOu9RTLaIJlqhtX2VnCpsI7xCUmjvDW9BbYTJ+sXOVKTS
          PQ52IZO//WVOBwDsGRrVPPtIW0ZCXjJpLn7JJdAa74TYpQssMLl/og5cPPUrPs15
          +KBI7ZF+7hT+fuz05FKxytFnmp8BAc4iXnXrcRVcNgN95y9wwym76xi9tJB3mk9l
          XS+OwQDhdGtw4HG8ShTr89vWwFgX/ks6sOa2wNbhLOX3oaS3GmhtLFsrhlKRPLWh
          wsYvxg+DYbEg2Bob3vHvb4HtttpVfJg6jQH+D7NIWSnskvHSU7QYoWW92JffgE43
          taEVqlgUtp+h/UM3Fi5PlvK3CeTLRbeLyYd36BYCz9wgxz95G3gSIigkOgCBfcLl
          QiWmgicBlODGyW//8uSk/1Fgt86YEycszlnKDD2E/l5/j1e37cvY+rFugsvDa1sH
        ]]>
      </cipherText>
    </secure-library>
  </functions>

  <program>
    <root-segment>
      <event>
        <starttime>
          <multiplyOperator>
            <literal> 0.5 </literal>
            <systemVariable name="cal.rabi.period"/>
          </multiplyOperator>
        </starttime>
        <use-function name="thirdparty-func-1"/>
      </event>
      <event>
        <starttime unit="us">
          <literal>15</literal>
        </starttime>
        <pmtMeasurement>
          <channel>pmtChannel1</channel>
          <resource name="counter1"/>
          <countTime units="ms">
            <literal>5</literal>
          </countTime>
        </pmtMeasurement>
      </event>
    </root-segment>
  </program>
</experiment>
```



Listing 4.1: Complete Encrypted Program

```
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  <resources>
    <pmt-counter>
      <id>counter1</id>
      <uuid>df271d2d-5c44-4f3d-9efc-7529839e8dbe</uuid>
    </pmt-counter>
  </resources>

  <headers>
    <function-header name="thirdparty-func-1"/>
  </headers>

  <functions>
    <function name="thirdparty-func-1">
        <event>
            <starttime type="relative">
                <literal units="us">5</literal>
            </starttime>
            <simpleLaserPulse>
                <duration>
                    <literal unit="us">5</literal>
                </duration>
                <channel>CoolingLaser1</channel>
            </simpleLaserPulse>
        </event>
    </function>
  </functions>

  <program>
    <root-segment>
      <event>
        <starttime>
          <multiplyOperator>
            <literal> 0.5 </literal>
            <systemVariable name="cal.rabi.period"/>
          </multiplyOperator>
        </starttime>
        <use-function name="thirdparty-func-1"/>
      </event>
      <event>
        <starttime unit="us">
          <literal>15</literal>
        </starttime>
        <pmtMeasurement>
          <channel>pmtChannel1</channel>
          <resource name="counter1"/>
          <countTime units="ms">
            <literal>5</literal>
          </countTime>
        </pmtMeasurement>
      </event>
    </root-segment>
  </program>
</experiment>
```



Listing 4.2: Complete Decrypted Program

### 4.2.2 Subfunction Expansion

Subfunction expansion flattens the execution action tags[4]. It is simply macro-expansion of the corresponding `<function>` body. Each `<useFunction>` tag is replaced with the body defined previously. Since this process occurs after decryption, there is no difference between encrypted functions from a secure library, or those from a user-generated process. The expansion is recursive, so the sequence compiler supports function-within-function semantics. This expansion is shown in Listing 4.3.

[4]Flattening refers to the non-recursive execution of nested sub-functions. Loops are not un-rolled, so the execution may not be linear.

```
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  <resources>
     <pmt-counter>
      <id>counter1</id>
      <uuid>df271d2d-5c44-4f3d-9efc-7529839e8dbe</uuid>
    </pmt-counter>
  </resources>
  <program>
    <root-segment>
      <event>
        <starttime>
          <multiplyOperator>
            <literal> 0.5 </literal>
            <systemVariable name="cal.rabi.period"/>
          </multiplyOperator>
         </starttime>
        <event>
          <starttime type="relative">
              <literal units="us">5</literal>
          </starttime>
          <simpleLaserPulse>
              <duration>
                  <literal unit="us">5</literal>
              </duration>
              <channel>CoolingLaser1</channel>
          </simpleLaserPulse>
        </event>
      </event>
      <event>
        <starttime unit="us">
          <literal>15</literal>
        </starttime>
        <pmtMeasurement>
          <channel>pmtChannel1</channel>
          <resource name="counter1"/>
          <countTime units="ms">
            <literal>5</literal>
          </countTime>
        </pmtMeasurement>
      </event>
    </root-segment>
  </program>
</experiment>
```

Listing 4.3: Complete Program With Sub-Function Expansion

### 4.2.3 Symbolic Algebra Solution

With a flattened action sequence, symbolic algebra can now be expanded and the use of calibration values can be solved. This removes references to algebraic expressions, and replaces them with `<literal>` references. This is shown in Listing 4.4.

```
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  <resources>
     <pmt-counter>
       <id>counter1</id>
       <uuid>df271d2d-5c44-4f3d-9efc-7529839e8dbe</uuid>
     </pmt-counter>
   </resources>
   <program>
    <root-segment>
      <event>
        <starttime>
          <literal unit="ns"> 2389 </literal>
         </starttime>
        <event>
          <starttime type="relative">
              <literal units="us">5</literal>
          </starttime>
          <simpleLaserPulse>
              <duration>
                  <literal unit="us">5</literal>
              </duration>
              <channel>CoolingLaser1</channel>
          </simpleLaserPulse>
        </event>
      </event>
      <event>
        <starttime unit="us">
          <literal>15</literal>
        </starttime>
        <pmtMeasurement>
          <channel>pmtChannel1</channel>
          <resource name="counter1"/>
          <countTime units="ms">
            <literal>5</literal>
          </countTime>
        </pmtMeasurement>
      </event>
    </root-segment>
  </program>
</experiment>
```

Listing 4.4: Complete Program With Symbolic Expansion

### 4.2.4 Relative Time Solution

To support loops and branching, from this point forward all start times are converted to from absolute time, measured from start of the experiment, to relative time, measured from the last event.

```
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  <resources>
    <pmt-counter>
      <id>counter1</id>
      <uuid>df271d2d-5c44-4f3d-9efc-7529839e8dbe</uuid>
    </pmt-counter>
  </resources>
  <program>
    <root-segment>
      <event>
        <starttime unit="ns" type="relative">
          <literal>7389</literal>
        </starttime>
        <simpleLaserPulse>
          <duration>
            <literal unit="ns">5000</literal>
          </duration>
          <channel>CoolingLaser1</channel>
        </simpleLaserPulse>
      </event>
      <event>
        <starttime type="relative" unit="ns">
          <literal>7611</literal>
        </starttime>
        <pmtMeasurement>
          <channel>pmtChannel1</channel>
          <resource name="counter1"/>
          <countTime units="ns">
            <literal>50000000</literal>
          </countTime>
        </pmtMeasurement>
      </event>
    </root-segment>
  </program>
</experiment>
```

Listing 4.5: Complete Program In Relative Time

### 4.2.5 Runtime Calculation

It is at this point that a worst-case execution time can be calculated. This is performed by integrating all relative times and action durations, with each loop unrolled. With this information, the sequence compiler may reject excessively long programs.

### 4.2.6 Storage Allocation

At this point storage for each resource is allocated. In particular, the measurement actions consume a resource by name, but the FPGA only refers to these by their location within a results First-In / First-Out (FIFO) buffer. The XML code is annotated with the absolute position of each resource, and given an ID value for later retrieval by the user.

```
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  <resources>
    <pmt-counter>
      <id>counter1</id>
      <uuid>b4d7782f-480c-4a76-91bc-e0699493bfce</uuid>
    </pmt-counter>
  </resources>
  <program>
    <root-segment>
      <event>
        <starttime unit="ns" type="relative">
          <literal>7389</literal>
        </starttime>
        <simpleLaserPulse>
          <duration>
            <literal unit="ns">5000</literal>
          </duration>
          <channel>CoolingLaser1</channel>
        </simpleLaserPulse>
      </event>
      <event>
        <starttime type="relative" unit="ns">
          <literal>7611</literal>
        </starttime>
        <pmtMeasurement>
          <channel>pmtChannel1</channel>
          <resource name="counter1"/>
          <countTime units="ns">
            <literal>50000000</literal>
          </countTime>
        </pmtMeasurement>
      </event>
    </root-segment>
  </program>
</experiment>
```

Listing 4.6: Complete Program With Symbolic Expansion

### 4.2.7 Generation of Branch Lookup Tables

Branch lookup tables are required any time a measurement is taken that will affect the next executed instruction. Branch tables are indicated in XML code by the use of `<decision>` blocks, where each `<condition>` represents a new section of the program counter[5]. Branch-

[5]Such measurement-based changes to the execution graph are a core component in quantum error-correcting codes.

ing is performed by a lookup table based on the classical pattern measured, and the executed commands in each branch have unique values for time and program counter than those of other branches.

Branch lookup tables are generated in two steps: first, the measurement (identified as a `resource`) is dispatched from the FPGA module that performed the measurement (TTL, PMT, CCD, etc). to all other modules[6]. Secondly, the new values of a program counter are generated for the commands in each branch. A new lookup table is generated for each `<decision>` tag, and for each measured resource. In this way, the same measurement can be used with different conditions, and the same conditions with the same measurement can be used at different times in a quantum program.

### 4.2.8 Opcode Generation

After the generation of the branching lookup tables, the program is ready to map the specific action events to the corresponding opcodes. Each FPGA functional module contains several execution engines as described in Section 3.3. For each engine, a sequence of op-codes maps to an action tag (or lack thereof, in the case of `NoOp`). The relative times calculated by the sequence compiler can now be mapped to the values of the program counter. Each action is either a `NoOp`, a looping construct, a branching construct, or a `SetValue` opcode. Since all execution engines follow the same execution graph, the `NoOp` opcode occurs frequently in a given execution engines when an action occurs on a diffent engine. In this way, all engines can refer to the same program counter value at the same time.

Table 4.1 shows the evolution of two different execution engines. The two controls, for `CoolingLaser1` and `pmtChannel1` follow the same program counter, and are driven from the same absolute time reference. Notice that when `CoolingLaser1` performs its on and off operations, `pmtCounter1` executes `NoOp` (do nothing). This separation keeps the two in synchronization.

[6]This must be performed in advance of the branch, but after the measurement is completed; there is latency, but it is deterministic.

| Abs. Time (ns) | Rel. Time (ns) | Program Counter | CoolingLaser1 Opcode | pmtChannel1 Opcode |
|---|---|---|---|---|
| 7 389 | +7 389 | 1 | `SetValue 1` | `NoOp` |
| 12 389 | +5 000 | 2 | `SetValue 0` | `NoOp` |
| 15 000 | +2 611 | 3 | `NoOp` | `SetValue 1` |
| 50 015 000 | +50 000 000 | 4 | `NoOp` | `SetValue 0` |

Table 4.1: Opcode Generation. Inside each execution engine is a list of opcodes and execution times. At each time instant, the opcode is executed, resulting in a change to a parameter (such as on/off state). The program counter is a conceptual index into the list, allowing loops and branching logic. In a given quantum program, all execution engines in the entire FPGA network share the same execution times and program counters, but the opcodes are different for each engine.

### 4.2.9 Permission Validation

After Opcode generation, the quantum program is essentially complete. It is at this point, the program undergoes a permission validation step. Consistent with the idea of RBAC, certain controls are not available to normal users, such as the use of the ablation laser. However, when operating in the calibrator role, these controls become available. For some types of permissions, the information required is only know after the full program is compiled. For example, special permissions (described in Subsection 4.3.2) may be granted to allow very long programs. After permission validation, the program either is accepted or returned with an error to the user.

If the program is accepted for execution, the individual opcodes are sent to the FPGAs via the individual Linux drivers. Once opcodes are installed in each FPGA, they await the experiment start trigger. Upon completion the same FPGAs await download of the resulting measurements.

## 4.3 Experiment Scheduler

QuantumIon is designed to be, first and foremost, a shared resource that allows many interleaved users. As a result, the exclusive access to the trap, its configuration, and associated controls that is often enjoyed by experimentalists has been changed from tradition. In order to serve many remote users, a fair scheduling algorithm is crucial. At first glance, a

first-come, first-served algorithm seems appropriate, however, reconfiguration of some trap parameters may be expensive, in terms of the lifetime of the apparatus, or the reliability of results.

Instead, QuantumIon's scheduler is based of least-disruptive, first-served ordering with a maximum wait-time. The sequence compiler can determine what general alterations of the QuantumIon apparatus will occur over the course of any user's experiment, and also knows the basic state of the system at the end of each experimental run. Therefore, an experiment can be chosen from the queue that requires the least set-up cost.

Examples of costly set-up changes might be a change to the number of ions in the trap, changes to magnetic field over the course of the experiment, or shuttling of ions (which may be likely to eject the ion, causing the system to re-load ions into the trap).

### 4.3.1 Standard Scheduling

Consider the cost function $C_i$ for the $i^{th}$ experiment in the queue,where $\rho_i$ is the position in the queue of $N$ experiments, $\tau_i$ is the time an experiment has been waiting in the queue, and $\epsilon_i$ is the set-up cost of changing the trap state.

$$C_i = \alpha_\rho \frac{\rho_i}{N} - \alpha_\tau \frac{\tau_i}{\tau_{max}} + \alpha_\epsilon \frac{\epsilon_i}{\epsilon_{max}}. \tag{4.1}$$

$$\text{Choice} = \arg\min_i C_i \tag{4.2}$$

The corresponding weights $\alpha_\rho$, $\alpha_\tau$, and $\alpha_\epsilon$ control the importance of each factor. The next experiment is chosen by re-scanning the queue and choosing the lowest cost $C_i$ according to Equation 4.1. After the execution, the experiment is removed from the queue and a new calculation begins.

The cost function in Equation 4.1 gives the intuitive result for several scenarios. Assume that the weights are such that $\alpha_\epsilon > \alpha_\rho > \alpha_\tau$. In the simplest case, where no experiment has ever been deferred (that is, wait times $\tau_i$ monotonically decrease with queue position $\rho$), and all experiments are equally costly. In this case, the cost function becomes first-come, first-served since the position $\rho$ is dominant. A simulation of such a run is shown in the beginning, $t < 3$ sec, of Figure 4.3. In this case the scheduler consistently picks the first item in the queue (index one).

A more complex example considers an experiment that requires a change to the number of ions. This would be an increased equipment cost $\epsilon_i$, and would cause that experiment's deferral if another, less costly experiment is available, even if it was queued later. A

simulation of such a run is shown in the second part, $t \approx 3.75$ sec, of Figure 4.3. The scheduler begins deferring the program at the head of the queue in favor of the next (index two), to balance the set-up cost.

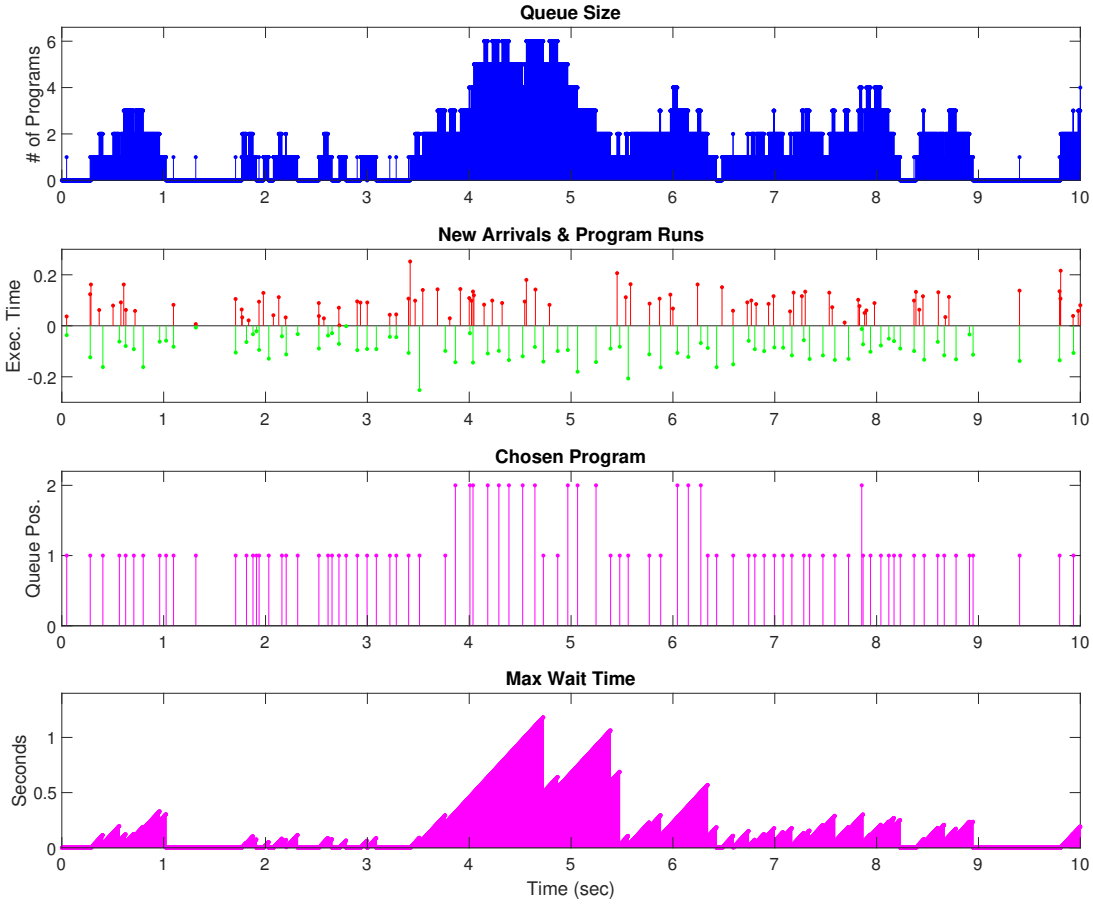


Figure 4.3: Simulation of standard scheduler. (Top) Queue size over a program evolution, showing increase as new programs arrive, and decrease as programs are executed. (Second) Recording of the events for new arrivals (positive) and executions (negatives), where the height is the program run time. (Third) The program position that is chosen for each run; most cases the front of the queue (index one) is chosen, with occasional deferment to later programs with better cost $C_i$. (Bottom) Maximum wait time in the queue, showing when deferment is overridden by excessive wait (near $t = 4.6$) sec.

The deferment has limitations, however. Even though it has been deferred, the costly experiment would be accumulating wait time $\tau_i$, which becomes significant after some time $\tau_{max}$. The $\alpha_\tau$ weight has the effect of forcing priority to experiments that have been deferred often so they eventually become queued. Figure 4.3 shows deferment is overridden by the fact that $t_{max}$ has been exceeded and the deferred program is run at approximately $t = 4.6$ sec. Again, the interplay of costliness $\epsilon_i$ and wait time $\tau_i$ work against each other. Costly changes (e.g. magnetic field and shuttling together) will produce more deferments, before the wait-time criteria wins out.

It is also noteworthy that Equation 4.1 does not attempt to defer beyond the current queue. Such attempts to predict what kind of experiments will come next are difficult, to say the least. As a result, when no other experiments are available, even the costliest one will be performed.

### 4.3.2 Scheduling for Special Experiment Runs

It is understood that some experiments may require a slight change to the algorithm described above. For example, a Ramsey interferometry experiment [38] may require several minutes to perform precision spectroscopy. There are many such experiments that could require exceptions to the basic queuing strategy described above. QuantumIon supports such requests, but users requiring such are given special, time-limited permissions for such circumstances[7].

Special access use cases that are envisioned include:

- Scheduling based on recency to a calibration parameter.
- Scheduling requests for back-to-back experiments.
- Scheduling requests for long-duration experiments.

As a special access, the scheduling algorithm of Equation 4.1 is modified. At the end of an experimental run, the queue is scanned first for special access experiments and the cost function is applied to these only. If none are ready, then the standard programs are run as normal. This has the effect of a priority queue, which is scheduled first, and then the standard queue. The priority queue concept is plagued with the possibility of denying

[7]Time-limited privileged access ensures that such experiments are possible, but such special access does not lock-out standard use for extended periods of time. As such, *experiment runs* are given special access, but there are no special *users*.

access to standard users, which is counter to the fundamental design of QuantumIon. As such, the special access programs should be exceptionally rare, and access should be granted with large periods of standard-only access.

The reader may note there is no provision for dedicated time-blocks as a special access, as is done for telescope time in astrophysical experiments. This is intentional, as it universally denies all other access, even when there is no experiment queued. It is believed that, overwhelmingly, most user needs are fulfilled with standard scheduling and occasional special access. Should long, dedicated time on an ion trap quantum computer be needed, it is likely this is a candidate for an entire dedicated machine for that experiment.

## 4.4 Execution Flowgraph

All quantum programs center around the idea of an Execution Flowgraph. This is the set of operations (be it laser pulses, measurements, etc) that become the basic operations that the control system must perform in order to satisfy the user's program. The primary purpose of the user program (see Chapter 6) is to describe such execution flowgraphs. Execution flowgraphs can be considered a form of directed graph.

The simplest execution flowgraph, the *flat* graph, is shown in Figure 4.4. In this example, a series of operations $\{a_1, \ldots a_n\}$ are executed in sequential order, ending with a measurement. There are no branches, nor partial measurements. This program runs from state preparation, through a series of gate operations, to a final measurement, whereby the program terminates. The final measurement is retrieved by the user at a later time, and the next quantum program in the queue (which may also be a calibration program) runs immediately afterward.

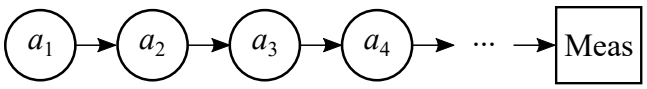


Figure 4.4: Simple Flat Execution Flowgraph. Each operation in a quantum program runs right after the other from beginning to end. There are no loops or branches. The experiment ends with a measurement. The execution times have been removed, but dictate when each operation $a_n$ will be performed.

A completely flat flowgraph, while conceptually simple, may be too verbose for practical use. For example, an operation may repeat hundreds of times in a non-trivial program.

User programs (see Chapter 6) can be generated in an external language such as Python or Matlab, and such repetitive operations can be generated programmatically, the problem of resource allocation within the FPGA may remain. the ability to *loop* over sections of the graph is such a common use case it should be incorporated into the QuantumIon system early. Figure 4.5 shows such a looping program. The operations $a_1$, $a_2$ are flat, while the operation $a_4$ loops $n$ times, and the segment $a_3$, $a_4$ repeats $m$ times before the final measurement.

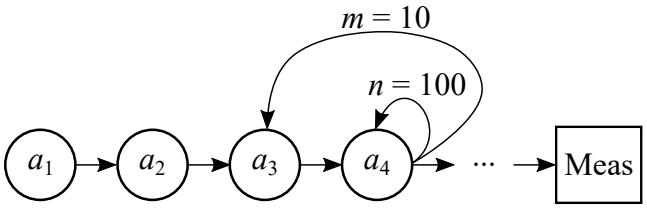


Figure 4.5: Execution Flowgraph with Looping. In this quantum program there are two loops, an inner one from $n = 1...100$, and an outer one from $m = 1...10$. The remainder of the program has no branches or further loops.

Quantum error correction requires the ability to change the future execution path based on a partial measurement of qubits in the ion chain. To support this operation, a decision block selects which path is chosen based on a lookup table. Each branch of the flowgraph is shown in Figure 4.6. The operation $a_n$ is chosen if the measurement encoding meets the lookup table definition, otherwise the operations $b_1...b_n$ is chosen.

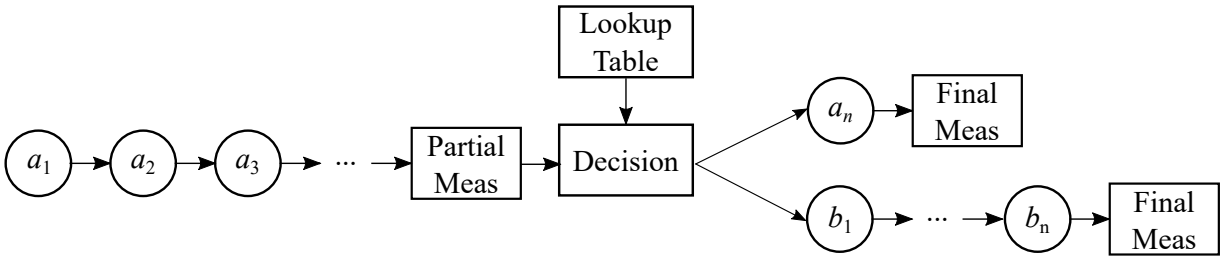


Figure 4.6: Execution Flowgraph with Decision Logic. The program begins with a simple linear execution, and then performs a measurement. A lookup table dictates which branch to take based on the measurement results. The decision block represents the time in which this decision is made.

Execution flowgraphs are executed within the execution engine of each FPGA module (see Section 3.3).

## 4.5 Calibration Database

The calibration database contains the latest values of calibration parameters. Calibration is a series of operations described in Chapter 8. As described in Subsection 1.5.2, the role of the Calibrator is to run quantum programs periodically and write to this database. The named values within are considered the latest values used by the symbolic algebra. Consider the pseudocode below for a $2\pi$-pulse:

```
program.addSteps(
    qi.SimpleLaserPulse(
        channel="raman1",
        tstart=1.0*qi.microseconds,
        duration=NamedConstant("cal.rabi.period")
    )
);
```

Listing 4.7: Simple Rabi Period

In this example, a pulse begins at time $t = 1.0\mu s$, and has a duration of one-half Rabi period. Since the Rabi period is a machine parameter (perhaps one that changes over time), the calibration database provides a mechanism for the user to defer calculation of the true value until just before the program executes. In doing so, the execution of the user's program always reflects the latest calibration.

Note that there is no explicit requirement to use the calibration database. Should the user have *a priori* information, or wish to perform a new, innovative type of calculation, the `duration` parameter in the example above can be manually entered as desired using the `<literal>` tag.

## 4.6 Symbolic Algebra Expansion

The use of the calibration database described in Section 4.5 is only of limited use by itself. The true power of the calibration database is the ability to chain multiple calibration values into the building blocks of various gates, i.e. to *parameterize* laser pulses. To this end, QuantumIon allows for the use of algebraic expressions in most controls. When combined with precision timing, and the calibration database, this results in an extremely powerful control language.

The symbolic package chosen for implementation in QuantumIon is the `GiNaC` package [39]. This package lends itself well to the QuantumIon project, since it is almost purely a calculation engine written as a C++ library.

Symbolic expressions are constructed in the QuantumIon XML language (Section 6.2). The basic building blocks are variables, constants, and transcendental functions. Custom mathematical functions are not supported, although the equivalent capability can be emulated using chains of the built-in functions. The variables are in the form of calibration database named constant[8]. When a new program is parsed the entire contents of the calibration database are imported as constants.

The code in Listing 4.8 shows the creation of a $\pi$-pulse. The Rabi period is a Named Constant, and the multiplication by a numeric constant turns the pulse duration into a symbolic expression.

```
program.addEvent(
    tstart=1.0*qi.microseconds,
    qi.SimpleLaserPulse(
        channel="raman1",
        duration=NamedConstant("cal.rabi.period") * 0.5
    )
);
```

Listing 4.8: Example using Symbolic Language in a Binding Language

The example in Listing 4.8 is written in a binding language. The important parts are the lines

$$\texttt{tstart} = 1.0\mu s \tag{4.3}$$

$$\texttt{duration} = \frac{\Omega}{2}. \tag{4.4}$$

Using the language bindings described in Section 6.8 and Section 6.9, the more verbose expansion into the XML language is as follows:

[8]The nomenclature in the XML language is called a *named constant*, although in the GiNaC language these are *variables*. The term *constant* implies that it doesn't change over the course of a quantum program, though its value is unknown.

```
<event>
  <starttime unit="us">
    <literal>1.0</literal>
  </starttime>
  <simplelaserpulse>
    <channel>"raman1"</channel>
    <duration type="expression">
      <productOperator>
        <literal>"0.5"</literal>
        <systemVariable>"cal.rabi.period"</systemVariable>
      </productOperator>
    </duration>
  </simplelaserpulse>
</event>
```

Listing 4.9: XML version of the symbolic language

An example of a more complex use of a calibration parameter is an expression relating a transcendental function of the Rabi frequency $\Omega$ with an applied polynomial in the intensity $I$

$$\texttt{duration} = \left(0.25 + 0.125I + 0.41I^2\right) \sin \frac{\Omega}{2}. \tag{4.5}$$

The corresponding XML code would be

```
...
<duration>
  <productOperator>
    <groupOperator>
      <sumOperator>
        <literal>"0.25"</literal>

        <productOperator>
          <literal>"0.125"</literal>
          <systemVariable>"cal.rabi.intensity"</systemVariable>
        </productOperator>

        <productOperator>
          <literal>"0.41"</literal>
          <powerOperator>
            <systemVariable>"cal.rabi.intensity"</systemVariable>
            <literal>"2"</literal>
          </powerOperator>
        </productOperator>
      </sumOperator>
    </groupOperator>

    <sineOperator>
      <productOperator>
        <literal>"0.5"</literal>
        <systemVariable>"cal.rabi.period"</systemVariable>
      </productOperator>
    </sineOperator>
  <productOperator>
</duration>
...
```

Listing 4.10: XML version of complex symbolic expression

## 4.7 Data Connection & Transport

All users are assumed to connect to QuantumIon via external network links. These links could be the world-wide-web, or from a local connection inside QuantumIon's own network. In either case, the main program must receive outside network connections and process them. The collection of protocols that accomplish this task is known as the *Transport* layer. The programmatic details are described in Subsection 4.1.2.

### 4.7.1 User Actions

In order to define the transport protocol, it is useful to define the basic actions, or Use Case, that the protocol would support. The high-level flow of information is shown in Figure 4.7. The user writes a program using one of the bindings described in Section 6.8 and Section 6.9. These bindings create the XML language syntax.

It is at this point where the transport layer begins. The user connects with the QuantumIon server, exchanging credentials as described in Section 4.1. This connection is established at the transport layer. Once logged-in (a successful connection), the user uploads the XML program to the server, along with any support resource files, such as waveform files for the AWG modules. The program must be queued and run, and the user can periodically check the status of their program. After completion, the QuantumIon physical apparatus saves the results of measurements into a long-term storage array. At this point, the user may download these results to their own computer for post-processing and analysis.

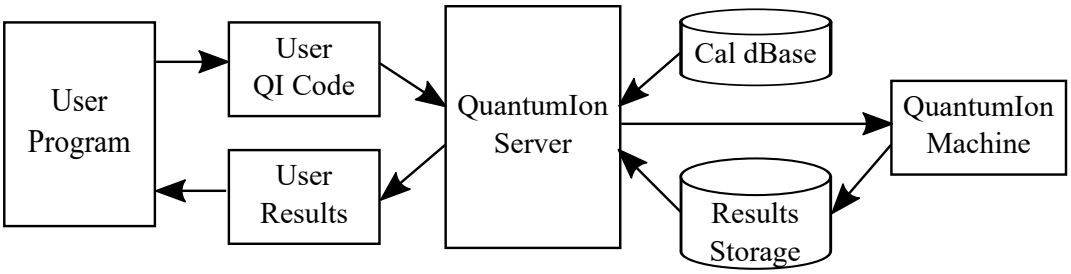


Figure 4.7: High-Level User Program Flow. The user program, written in a binding language, is converted to the QuantumIon XML language and sent to the server. The server converts this XML, with the help of the calibration database, into opcodes for the FPGAs that control the ion trap apparatus. Results of measurements are provided to the storage array. The user then requests these results resources after the program completes.

Figure 4.7 shows the need for a series of basic interactions between the end user, and the QuantumIon main server. These are summarized as follows:

- The ability to log-in and log-out of the system. This process establishes the current session and the permissions the user has, including which controls are allowed, and access to which files is permitted.

- The ability to upload a quantum program. This process effectively creates a quantum program ID that can be connected to other resource files, exposing, e.g. filenames, resource IDs and other shared information between separate parts of a whole experiment.
- The ability to upload resource files, such as waveforms for AWG modules. This allows customized waveforms created for special quantum gates to be used repeatedly.
- The ability to download recorded measurements. Again, the resource concept described in Section 6.4 is used to create IDs for these measurements, and once the program is finished, the user can retrieve them.
- The ability to request special queuing rights for certain programs. For example, an extra-long duration program is scheduled differently than the typical short-run program, so that as many short-run programs are completed prior to long-run ones.
- The ability to query the status of a program in the queue. This allows a user to hold off on requests for results until the program is complete, or to check the expected time until a program is run.

### 4.7.2 SOAP Protocol

Service-Oriented Application Protocol (SOAP) [40] is a standard protocol for web services throughout the internet. SOAP provides a mechanism for encapsulating commands (SOAP Remote Procedure Call (RPC)), or documents (SOAP Document Exchange), and transferring them over one of the lower-layer protocols commonly used in the internet. In the case of QuantumIon, SOAP is transported via standard HTTPS links[9][10]. Although HTTPS is technically the transport agent according to the traditional OSI reference model, we will generally refer to SOAP as a transport agent instead of the more clumsy *XML-RPC-over-HTTPS-with-SOAP-encapsulation*.

SOAP consists of the transmission of specially formatted XML messages, known as *envelopes*[41]. HTTPS provides a request-response mechanism.

[9]HTTPS is the secure version of the HTTP webpage protocol. HTTPS is chosen as it is compatible with most university campus firewalls.

[10]QuantumIon uses HTTPS, however the content is *not* Hypertext Markup Language (HTML). What is being transmitted is not webpage data, and has no visual representation.

```
<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
  <s:Header>
    <m:transaction xmlns:m="soap-transaction" s:mustUnderstand="true">
      ...
    </m:transaction>
  </s:Header>
  <s:Body>
    ...
  </s:Body>
</s:Envelope>
```

Listing 4.11: Skeleton of a SOAP envelope

The SOAP protocol is under active development as part of the construction of Quantum-Ion. At this time the following basic commands are defined:

- `login` Command
- `logout` Command
- `uploadProgram` Command
- `enqueueProgram` Command
- `uploadResource` Command
- `downloadResource` Command
- `checkQueueStatus` Command
- `addEncryptionKey` Command
- `removeEncryptionKey` Command

## 4.8 Conclusion

This chapter describes the software residing on the main Linux server. This software performs a number of important functions, particularly interactions with the user in the

form of security and communications, and the translation of XML quantum programs into the FPGA opcodes needed by the execution engines. One of QuantumIon's most powerful features, the symbolic algebra, allows users to abstract away machine-specific properties, and instead focus on the desired operations.

This chapter also describes the use of XML, the same language used for quantum programs, as a means for communicating, security, and starting/stopping programs, uploading waveforms, and retrieving results. In short, XML becomes the universal language for *everything* that happens around the QuantumIon computing ecosystem.

The previous chapter discussed the FPGA modules and how they provided precision timing to the hardware. The next chapter is devoted to a single, extremely important, FPGA module: the Arbitrary Waveform Generator (AWG) module. This module provides the power to fine-tune the shapes of laser pulses used in quantum operations on the ions in the trap. Experience has shown this to be a key ingredient to high-fidelity quantum gates, and also an active area of research to improved ion traps in the future.

# Chapter 5

# Arbitrary Waveform Generation

In Chapter 3, the FPGA modules were introduced, and the details of each module were described. One important type of FPGA module has such complexity, and is so important to a high-quality ion trap quantum computer, that it deserves its own chapter. The AWG module provides the main way to optimize the shapes of laser pulses. QuantumIon allows the user to define these pulse shapes directly, a feature not available in other quantum computing platforms.

The AWG feature provides a high-speed, highly configurable control of the detuning RF signal. Results such as those in [42] and [11] have shown considerable improvements in gate fidelity as the result of a shaped intensity, phase, or frequency profile on the addressed Raman beams. The pulse shape described in [42] required a global optimization of $N$ piecewise-constant steps to get best results. It is likely that pulse shapes will be an active field of research, and so a platform like QuantumIon is well-poised to support arbitrary pulse shapes.

## 5.1 Hardware Topology

The AWG hardware topology is shown in Figure 5.1. AWG use begins with a user-specified waveform[1]. Waveform samples are held in the *waveform storage* array; this specialized high-speed storage system communicates via the Fibre Channel network using a built-in transceiver. The users uploads waveforms he or she has generated via a standard Ethernet connection. It is the job of the main program to parse these files and upload them to

[1]Pre-made, optimized waveforms may also be available, but the concept does not change.

the waveform storage in the specialty filesystem (Section 5.3). When the user program is running, FPGA cards request blocks of samples from the waveform storage array, effectively streaming the waveform.

## 5.2 Fibre Channel Implementation

Fibre Channel (FC)[43][44] is a high-speed optical network designed specifically for low-latency, networked storage[2]. Unlike general-purpose protocols such as Ethernet, Fibre Channel is a specialized network protocol. The network topology is shown in Figure 5.1. Each FPGA contains the ADC/DAC that form the voltage generation, as well as a transceiver slot, labeled as QSFP+, for high-speed optical connection to the switch.

[2]The term and spelling refers to the two-dimensional connected-ness of the network, reminiscent of a cloth weave with fibre filaments as the thread. Both copper- and fibre-optic connections exist.

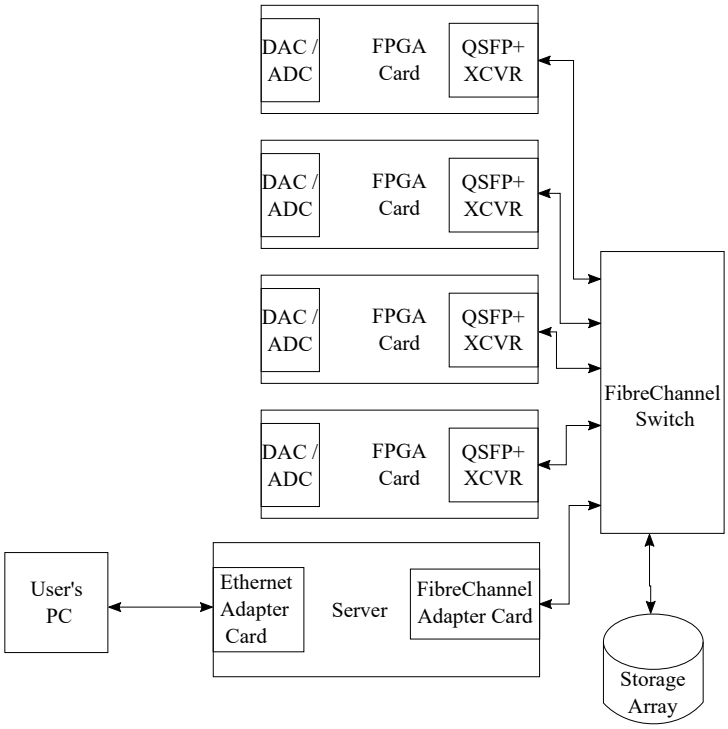


Figure 5.1: Fibre Channel Network. The AWG provides the ability to stream extremely long waveforms to the resulting FPGA hardware via a high-performance storage network. Each FPGA is directly connected to the hard drives, without the need of an intermediate server as a bottleneck. The server also independently connected to the drives in order to upload waveforms, and retrieve results.

Fibre Channel provides a convenient platform for the streaming of samples for several reasons:

- *Scalable network*
- *Low-level access*
- *High speed, low latency*
- *Comparison against similar technologies*

### 5.2.1 Scalability

As seen in Figure 5.1, the overall ability to generate arbitrary voltage waveforms at high speed is distributed across several FPGAs. Each FPGA is responsible for up to four individually-addressed Raman beams within the trap, and so scalability of the number of qubits will require scalability of the FPGAs. It is also reasonable to expect that the size and complexity of the users' stored waveforms will grow over time, requiring a storage array that can be easily extended[3]. A switched-fabric network provides such scalability; by imposing a network structure in the core design, new FPGA cards are simply added to the FC switch, and new storage arrays (or drives within a single array) are done likewise.

### 5.2.2 Low-level Access

A second requirement, that of low-level access, is necessary for speed requirements as detailed in Section 5.3. In this case, low-level access means access to block- or sector-level commands within each hard drive. The Fibre Channel Protocol is one application layer protocol within the FC stack[4] Low-level access takes the form of SCSI block commands, and can be transmitted and received by an FPGA without the need for an operating system.

### 5.2.3 Latency and Speed

The third requirement, high-speed and low-latency, comes from the need to stream data to the FPGAs at such high speeds. Consider the example below, streaming 100 microseconds

[3]This rules out simpler topologies, such as directly connecting one hard drive per FPGA

[4]Fibre Channel (FC) refers to the networking and physical layers, but can be easily confused with *Fibre Channel Protocol* (FCP), the specific encoding of SCSI block commands onto the FC network. Rarely is FC used without FCP, and rarely are encodings other than FPC found on a FC network, so FC and FCP can be used synonymously.

of real-valued 16-bit data on all sixteen AWG channels:

$$\begin{aligned} F_{sample} &= 2 \times 10^9 \quad \text{Samp/sec} \\ N_{bits} &= 16 \\ N_{channels} &= 16 \\ T_{burst} &= 10^{-4} \quad \text{sec} \\ R_{data} &= F_{sample} \times N_{bits} \times N_{channels} \\ &= 512 \quad \text{Gbit/sec} \\ N_{burst} &= R_{data} \times T_{burst}/8 \\ &= 6.40 \quad \text{MByte/burst.} \end{aligned} \tag{5.1}$$

Of importance in this example is the total network data rate $R_{data} = 512$ Gbit/sec . Such throughput is unrealistic in even the most advanced IT environments, even though the actual transmitted data is modest. However, Equation 5.1 assumes all data is flowing through a single "fat pipe". In the single-array topology of Figure 5.1, the storage array must in fact operate at this speed. However, each FPGA is only responsible for its four channels, requiring only 128 Gbit/sec. This speed is satisfied by latest-generation FC interfaces (so-called Gen-6 fibre channel) [5].

It is noteworthy that this is a worst-case calculation, assuming full streaming of all channels for an indefinite amount of time. In reality, gates (and therefore AWG waveforms) are expected to be reused, with gaps between them. These gaps can be considered as a duty cycle $R_{duty}$ In such cases, the effective throughput $R_{data,\text{eff}}$ is

$$R_{data,\text{eff}} = F_{sample} \times N_{bits} \times N_{channels} \times R_{duty}. \tag{5.2}$$

### 5.2.4 Comparison Against Similar Technologies

There are several physical-layer network protocols that support the above requirements, in addition to Fibre Channel. Among the most important ones are Ethernet, Infiniband, and PCI express. In this section, the implementation difficulties are discussed. Particular emphasis is placed on the implementation of these protocols in FPGA devices.

PCI Express is the current de-facto standard for motherboard backplanes. Current generation (Gen-4) [45] specifies 16 billion transfers per second, approximately 64 GB/sec. PCIe does not support a storage format directly, however it is capable of interfacing to

[5] Early versions of QuantumIon are likely to use lower-speed FC interfaces. Of importance here is that current technology already exists to support the required speeds.

storage arrays through Host Bus Adapter (HBA) cards. It is natively supported by FPGA vendors with a great deal of support. PCIe is used in QuantumIon as the main server-to-FPGA interface for configuration and programming, as described in Subsection 3.2.3. However, PCIe cannot be easily scaled to the dozens of FPGAs necessary for a multiple-trap network; such would require server motherboards that simply do not exist. It is therefore not viable for a storage strategy.

Infiniband is another high-speed protocol and is discussed in Section 3.2. It has a low-level disk access layer in the form of SCSI RDMA Protocol (SRP)[46]. However, at present the implementations require heavy operating system support (such as Linux `lio`). As such, SRP is difficult to implement directly on FPGAs, and no SRP-based storage array hardware was found in industry[6]. Infiniband is used in QuantumIon as the message-passing technology for sharing partial measurements, see Subsection 3.2.4.

Ethernet is by far the most popular and well-understood of the network physical layer protocols. At its fastest implementations, Ethernet can meet the demands of Equation 5.1, and is the most cost-effective solution. Ethernet is used elsewhere in QuantumIon, as the main connection to the remote user (see Figure 2.6). It provides two network-based protocols for storage: iSCSI [47], and FCoE[48].

iSCSI provides a full 'Internet-ready' implementation of the SCSI command set, using standard routers and switches; it is also supported on most storage drive arrays and operating systems. However, iSCSI is based on the full TCP/IP protocol, meaning it allows for lost packets, fragmented packets, route discovery, and a full security layer. The full TCP/IP stack must be implemented on each FPGA in order to access the iSCSI array. Unfortunately, TCP in particular [49], is designed around heavy use of dynamic memory, packet reassembly, and other concepts that are easily coded in a high-level language such as C or Python, but difficult in a HDL for FPGA use.

Fibre Channel over Ethernet (FCoE) is the other major implementation of a storage network built on the Ethernet physical layer. In FCoE, the Fibre Channel Protocol is used as the command set, however the physical layer of FC is not. Essentially, the FCP messages ride over normal Ethernet wiring, but the familiar TCP/IP routing is not used, which simplifies FPGA coding. This has only a partial advantage: cabling is simpler by using normal Category 6e network wiring. However, Ethernet was designed to support a hostile network environment[7], and as such switches support heavy buffering, retries, and other strategies to ensure reliable data delivery. The Fibre Channel Protocol has

[6]Several storage arrays with dedicated Linux servers that implement SRP were found, but discarded from consideration.

[7]Packet loss and fragmentation, as well as long latency.

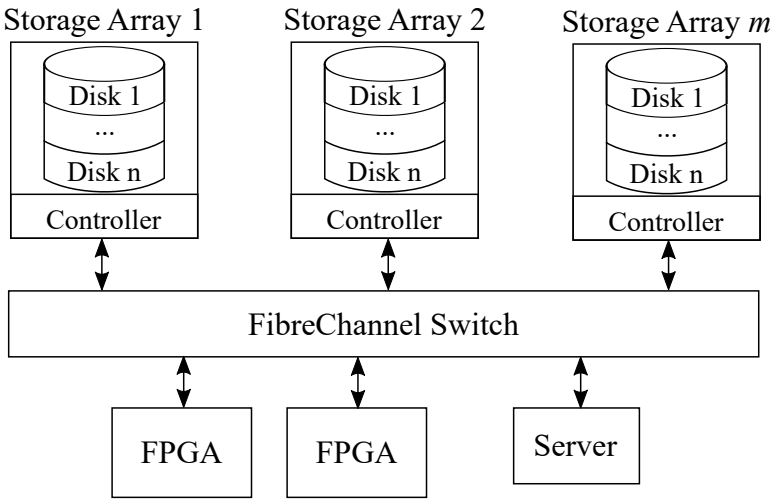


Figure 5.2: Parallel Storage Array. To further remove bottlenecks, the storage arrays are comprised of individual addressable hard disks. When a chunk data from one disk is being accessed, the next chunk can be pre-fetched on another drive. The concept can be further extended to multiple arrays to further reduce the amount of traffic that any one array controller must manage.

its own mechanisms for this, which conflict with those of Ethernet. As a result, FCoE requires specialized switches that generally can't coexist with other general-purpose traffic. Similarly, the storage array itself must support FCoE. Only one major array vendor was found, which was also prohibitively expensive.

Based on these requirements and the particulars of each protocol, the use of the Fibre Channel network and a corresponding FC storage array is the best selection.

## 5.3 Fast Storage Array Filesystem

In Subsection 5.2.3 it was noted that the speed of the AWG network effectively 'bottlenecks' at the storage array. There are two strategies, when combined, can alleviate this problem: storage arrays with parallel disks, and a distributed filesystem.

A parallel storage system is shown in Figure 5.2. In this topology individual arrays are connected to the FC switch. If the waveform files are appropriately distributed into blocks

throughout the $m$ different arrays, each array need only push data at

$$R_{array} = \frac{R_{data}}{m} \quad \text{GB/sec.} \tag{5.3}$$

A single drive retrieving and transferring data at $R_{data}/m$ is still infeasible; latencies associated with locating the drive sector, head, and cylinder, for a particular block of data can be significant. To achieve streaming, each disk must be able to deliver samples at the $R_{data}/m$ rate. In reality, each storage array is not a single drive, but a collection of $n$ individual hard drives. A *controller* within each array receives the FC messages and distributes them to the corresponding drive. As with array-level parallelism, distributing the waveform file within each disk, within each array, further reduces the demand for data throughput. This is achievable since the controllers' latencies are much shorter than the access time for the drives themselves. Now the rate for each of $n$ disks, if the file is properly distributed, is

$$R_{disk} = \frac{R_{array}}{n} = \frac{R_{data}}{nm} \quad \text{GB/sec.} \tag{5.4}$$

Thus, a modest arrangement of eight arrays, each with eight drives yields a per-device throughput of

$$R_{data} = 512 \quad \text{Gbit/sec} \tag{5.5}$$

$$R_{array} = 64 \quad \text{Gbit/sec} \tag{5.6}$$

$$R_{disk} = 8 \quad \text{Gbit/sec} \quad = 1 \quad \text{GB/sec.} \tag{5.7}$$

Note that this distributed waveform file concept must be *disassembled* to each array and disk from the single source file, and then *reassembled* during the experiment. In order to perform these two tasks, the low-level control of the drives indicated in Subsection 5.2.2 becomes apparent. Were a single server responsible for these two tasks[8], such a server is now the bottleneck. Instead, with low-level control the server need only place each block of samples in the appropriate drive/array pair[9], and communicate this to the FPGAs. Upon playback within a quantum experiment, each FPGA requests the blocks that make up the waveform by its drive array coordinates. Since the server communicates directly to the array only when the file is stored, this lengthy process happens before the experiment is run. Correspondingly each FPGA only requests the data blocks it needs (not those of other FPGAs).

[8]This is the case for the Network File System and Samba protocols, two popular remote storage systems.

[9]Hard drives actually contain a third ordinate, the *sector*, which locates a position on the drive. This is merely another layer of the same hierarchy.

The layout of such a filesystem is shown in Table 5.1, which illustrates the *disassembly* process. In this example, four arrays of four disks each are populated with four independent files. Then Table 5.1 is what the main server distributes files `F0-F3` into small blocks on each of the arrays. Notice the (Array,Disk,Sector) = (0,0,0) coordinate contains the first block of the first file.

| | | Array 0 | | | | Array 1 | | | | Array 2 | | | | Array 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Disk | | | | Disk | | | | Disk | | | | Disk | | | |
| | | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| Sector | 0 | F0,0 | F0,4 | F0,8 | F0,12 | F0,1 | F0,5 | F0,9 | F0,13 | F0,2 | F0,6 | F0,10 | F0,14 | F0,3 | F0,7 | F0,11 | F0,15 |
| | 1 | F1,0 | F1,4 | F1,8 | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| | 2 | F2,0 | F2,4 | F2,8 | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| | 3 | F3,0 | F3,4 | F3,8 | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

Table 5.1: File System Geometry. Four files, `F0-F3`, are partitioned into the sectors of an array of drives. Any given file first spans array-wise, then disk-wise. Multiple files are offset by the drive sectors. The $b^{th}$ block of the $n^{th}$ file is labeled as `Fn,b`

The second block is located on Array 1 at (1,0,0), then (2,0,0), and so on. When the last array is reached, the following block moves back to Array 0, but changes to disk 1, i.e. (0,1,0). The second file begins at the second sector of the first array, first disk (0,0,1), and then spans each Array in order. While the numbers of files are arbitrary, the position and layout is required for a simple implementation on FPGA hardware.

The file layout iterates as shown in Table 5.2, where the evolution of coordinates over each file is made clear. The location of blocks for one entire file is given by one columns of this table. The main program sends this column to the corresponding FPGA as part of setup for the quantum program. At the appropriate trigger time, the FPGA then sequentially reads these blocks from the arrays, thus providing the waveform samples necessary for playback.

| Block $b$ | (Array,Disk,Sector) File `F0` | File `F1` | File `F2` | File `F3` |
|---|---|---|---|---|
| 0 | (0,0,0) | (0,0,1) | (0,0,2) | (0,0,3) |
| 1 | (1,0,0) | (1,0,1) | (1,0,2) | (1,0,3) |
| 2 | (2,0,0) | (2,0,1) | (2,0,2) | (2,0,3) |
| 3 | (3,0,0) | (3,0,1) | (3,0,2) | (4,0,3) |
| 4 | (0,1,0) | (0,1,1) | (0,1,2) | (0,1,3) |
| 5 | (1,1,0) | (1,1,1) | (1,1,2) | (1,1,3) |
| 6 | (2,1,0) | (2,1,1) | (2,1,2) | (2,1,3) |
| 7 | (3,1,0) | (3,1,1) | (3,1,2) | (4,1,3) |
| 8 | (0,2,0) | (0,2,1) | (0,2,2) | (0,2,3) |
| 9 | (1,2,0) | (1,2,1) | (1,2,2) | (1,2,3) |
| 10 | (2,2,0) | (2,2,1) | (2,2,2) | (2,2,3) |
| 11 | (3,2,0) | (3,2,1) | (3,2,2) | (4,2,3) |

Table 5.2: Block Coordinates in Filesystem. For each of the four files shown in Table 5.1, the contents are played-back by going down each column and retrieving the coordinates. The ordering of (Array, Disk, Sector) is identical to that of Table 5.1.

## 5.4 Conclusion

This chapter described the primary laser pulse-shaping capability of QuantumIon: the ability for a user to generate arbitrary pulse shapes for the control of each ion. However, the flexibility offered by QuantumIon requires a great deal of digital engineering, including an extremely high-speed network backbone, customized protocols, and a customized file system. As such, the design of the AWG module borrows heavily from supercomputer engineering, radar system engineering, and the design of computer datacenters.

The next chapter introduces a feature that is perhaps most interesting to the end user: the language used to describe quantum programs. This is the only means in which QuantumIon interacts with the outside world. The XML language is the lowest level of control, and so the next chapter will describe all the operations of quantum programs in detail. However it will be shown that the end user may actually describe programs in any one of several familiar high-level languages.

# Chapter 6

# User Language

In Chapter 4, the main program was described along with one of its core functions: the compiling of the users' quantum programs into the FPGA timing instructions that control the hardware. In this chapter, the format of these quantum programs is described.

QuantumIon takes a layered approach to these program: there is a low-level language suitable for machine compilation, and several higher-level languages that the user may choose from. The bulk of this chapter will be concerned with the lowest-level language: the eXtensible Markup Language (XML). Because the low-level XML is intimately tied to the control system itself, it is the area of most focus in this thesis, and its format may seem uncomfortably verbose to the user. Fortunately, users are not expected to actually program in XML. At the end of this chapter, the details of how more familiar high-level languages (such as Python) are to be implemented through the use of *binding*.

A major innovation of QuantumIon is how it supports realtime decision logic directly at the level of precision-timing provided by the FPGA modules. This branching logic is also described in detail in this chapter.

## 6.1 Rationale

The user language is the primary way in which a user describes quantum programs. From this perspective, the user language is QuantumIon. As a result, the user programming language must be accessible, usable, and friendly to a wide range of researchers, graduate students, and professors.

Computer languages progress rapidly, gain favor (sometimes briefly) and are subsequently retired over time. Popular high level languages include FORTRAN, Perl, Java, C++, Python, Rust, and Go [50]. Unfortunately, the design of QuantumIon's core language cannot change so quickly without internal redesign. To provide flexibility and so-called future-proofing of the system, the user language is separated into two parts: a static intermediate language (which may be unfriendly, but should be very expressive), and a series of *Language Bindingss* to more popular languages such as Python, Matlab, and others. This section focuses on the intermediate language, a language template in XML, and bindings in two popular languages for quantum computing: Python and Matlab. It is hoped that by designing around these two languages simultaneously, portability to other languages is easy.

## 6.2 XML Intermediate Language

The eXtensible Markup Language (XML) is a is a tag-based language that is extremely popular in computer communications such as the world-wide web. XML is highly machine-readable, and represents structured data, and list data very well. A particular layout of tags is known as the XML *schema*. XML parsers by default ignore unknown tags, allowing future expansion without redesigning the schema. Additionally, several mature parsers are written as libraries for C++[1]. For these reasons, the XML language is used as the native language that the user code ultimately represents to QuantumIon.

XML uses the concept of *tags* to delineate different elements of the structured data. There are three types of tags: the start and end tags, and the empty-element (self-closing) tag. The start and end tags have the syntax `<tag-name>` and `</tag-name>` respectively, where *tag-name* must be the same for each. The start and end tags form a pair that may contain other tags or data. The empty-element tag, as described by the `<tag-name/>` cannot contain data. Start- and empty-element tags can contain *attributes* (end tags cannot) of the form `<tag-name attribute="attrib value">` or `<tag-name attribute="attrib value"/>`.

The descriptions below detail each of the major structural tags that represent the program. At first glance, it may seem that there is excessive use of tags leading to an unnecessarily verbose programming language. However, it should be recalled that the XML language is not designed to be efficiently coded by humans; instead the intent is to provide a schema that is easily generated by true user-oriented languages like Python.

[1] QuantumIon uses the Xerces parser

### 6.2.1 Experiment Tag

The quantum program body is contained within the `<experiment>` tag. Within this tag, the XML namespace `https://iqc.uwaterloo.ca/quantumion`[2] This root container encapsulates all other parts of the QuantumIon XML code.

```
<?xml version="1.1" encoding="UTF-8" standalone="no" ?>
<experiment xmlns:qi="https://iqc.uwaterloo.ca/quantumion">
  ...
</experiment>
```

Listing 6.1: XML Outer Container

### 6.2.2 Resources Tag

Resources such as PMT counters, CCD camera images, and AWG waveforms are contained within the `<resources>` tag. Three types of resources are defined: counters, images, and waveforms. For each resource, a unique identifier number is created by the user (i.e. the language binding). This ID is used later when the resource is attached to a measurement. Additionally, all resources have an optional `name` field, which defaults to the ID if absent. The name may be used by the user to identify the resource in some human-readable format. The name is preserved when the user downloads the results, but is otherwise not used.

[2]The URL in the namespace is nothing more than a string that ties the tags used in one XML schema to some signature. It needn't be a website with any special significance.

```
<experiment xmlns="https://iqc.uwaterloo.ca/quantumion">
  <resources>
    <awgWaveform type="file" filename="mine.mat">
      <id>"12345"</id>
      <name>"my awg waveform"</name>
        </awgWaveform>
    <ccdImage type="image">
      <id>"34567"</id>
      <!-- name field defaults to "34567" -->
    </ccdImage>
    <pmtCounter type="integer">
      <id>"45678"</id>
      <name>"counter number 1"</name>
        </pmtCounter>
    <pmtCounter type="integer">
      <id>"56789"</id>
      <name>"counter number 2"</name>
    </pmtCounter>
  </resources>
</experiment>
```

Listing 6.2: XML Resources example

### 6.2.3 Program Tag

The `<program>` tag defines the body of the program. The basic philosophy is that of a directed graph with loops. The primary element is the `<segment>` container tag. The starting point of the quantum program is the `<root-segment>` tag. Segments are delineated using `<decision>` blocks.

```
<experiment xmlns="https://iqc.uwaterloo.ca/quantumion">
  <program>
    <root-segment>
      ...
      </root-segment>
    </program>
</experiment>
```

Listing 6.3: XML Program Example

### 6.2.4 Decision Tag

The decision block represents the comparison of a classical measurement, whose result changes the active segment. For example, a typical quantum program might consist of a set of preparation steps (the root segment), a measurement of an error syndrome, and the application of either error correction (segment 1) or computational gates (segment 2). The measurement of some ancilla qubit[4] forms the basis for which segment is to be taken. The preparation and measurement form action tags (see below), and the root segment ends with a decision block.

Decisions are tied directly to a previous measurement. As noted previously, measurements require the definition of resources for identification. In the course of defining the measurement resource, parameters such as thresholds are defined, so the measurement resource results only in a measured state.

Decision blocks are marked with the `<decision>` tag, and the sensor used to make this decision is marked with the `resources` attribute. Each condition is marked with the `<condition>` tag and the measured `state` attribute, or `x` for a *don't-care* state. Don't-care states drastically reduce the size of the lookup tables, and are provided in the event that large programs push up to QuantumIon's memory limits. Listing 6.4 shows a two-element decision block utilizing only the first channel of a previous measurement.

```
<experiment xmlns="https://iqc.uwaterloo.ca/quantumion">
  <program>
    <root-segment>
        ...
        <!-- some measurement corresponding to measurement-1 -->
        <decision resource="measurement-1">
          <condition state="xxxx_xxxx_xxxx_xxx0">
            ...
          </condition>
          <condition state="xxxx_xxxx_xxxx_xxx1">
            ...
          </condition>
        </decision>
    </root-segment>
  </program>
</experiment>
```

Listing 6.4: XML Decision Example

### 6.2.5 Segment Tags

Each sequence of actions between decision blocks is surrounded by `<segment>` tags[3]. Segments form the basic building blocks of a linear sequence of operations. The root segment is contained with the outermost `<program>` tag. Subsequent `<segment>` tags are contained within the condition blocks of subsequent decision blocks.

```
...
<root-segment>
  <event>
    <decision ...>
      <condition ...>
        <segment>
          ...
        </segment>
      </condition>
      <condition ...>
        <segment>
          ...
        </segment>
      </condition>
    </decision>
  </event>
  ...
<root-segment>
```

Listing 6.5: Segment Tags

### 6.2.6 Event Tags

A segment is composed of `<event>` tags. Each event has a start time tag, `<starttime>`, which may be a symbolic algebra expression or literal. Within the event tag is one or more actions that are to be performed at that time. The start time has optional `units`, and `type` attributes. The `unit` attribute specifies a time units of `ns`, `us`, `ms`, `sec`. The `type` attribute is used to define relative time vs. absolute time.

[3]The `<root-segment>` is a type of segment tag.

```
...
<root-segment>
  <event>
    <starttime unit="us">
      <literal>50></literal>
    </starttime>
    ...
  </event>
  <event>
    <starttime unit="us" type="relative">
      <literal>5></literal>
    </starttime>
    ...
  </event>
  <event>
    <starttime unit="us" type="relative">
      <literal>5</literal>
    </starttime>
    ...
  </event>
  ...
<root-segment>
```

Listing 6.6: Event Tags

### 6.2.7 Action Tags

Within each event tag is several *action* tags[4]. Action tags are most of the actual electronic controls, such as laser pulses and measurements. Note that some actions are transient, i.e. the `<simpleLaserPulse>`, and others are permanent (until further change), like `<setMagField>`.

[4]The name action tags is descriptive only. No tag actually has this name.

| Description | XML Tag |
|---|---|
| No operation | `<noOp>` |
| Laser Pulse | `<simpleLaserPulse>` |
| AWG Laser Pulse | `<awgLaserPulse>` |
| Change Magnetic Field Setpoint | `<setMagField>` |
| Change Trap DC Electrode | `<setDCElectrode>` |
| Change Polarization | `<setPolarization>` |
| Change DDS Frequency Setpoint | `<setDDSFrequency>` |
| Change DDS Amplitude Setpoint | `<setDDSAmplitude>` |
| Change DDS Phase | `<setDDSPhase>` |
| CCD Measurement | `<ccdMeasurement>` |
| PMT Measurement | `<pmtMeasurement>` |
| Single TTL Input | `<ttlMeasurement>` |
| Change Single TTL Output | `<setTTLValue>` |
| Change PID Gains | `<setPIDcoefs>` |

Table 6.1: Action Tags. These XML tags are used in quantum programs to dictate changes to parameters. Each tag corresponds to a `SetValue` opcode on a particular FPGA execution engine. The `<noOp>` tag is the obvious exception.

#### 6.2.7.1 NoOp Tag

The `<noOp>` tag performs no operation. It is often used to consume delays using `<event>` tag with relative time. The `noOp` tag takes no additional arguments.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <noOp/>
</event>
...
```

Listing 6.7: NoOp Example

#### 6.2.7.2 simpleLaserPulse Tag

The `<simpleLaserPulse>` action defines a fixed-duration on-off laser pulse of a specified duration. It requires a `<channel>` tag indicating the appropriate laser, and a `<duration>` tag with an expression for the on time. The laser on time is the start time of containing event.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <simpleLaserPulse>
    <channel>"coolingLaser1"</channel>
    <duration units="us">
      <literal>"15"</literal>
    </duration>
  </simpleLaserPulse>
</event>
...
```

Listing 6.8: simpleLaserPulse Example

#### 6.2.7.3 awgLaserPulse Tag

The `<awgLaserPulse>` action defines the playback of an AWG modulated laser pulse of a specified duration. It requires a `<channel>` tag indicating the appropriate laser, and a `<resource>` tag defining the playback filename. The playback begins at the start time of containing event. The file will play to completion.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <awgLaserPulse>
    <resource name="file1234"/>
    <channel>"ramanLaser1"</channel>
  </awgLaserPulse>
</event>
...
```

Listing 6.9: awgLaserPulse Example

#### 6.2.7.4 setMagField Tag

The `<setMagField>` action defines a change of the trap magnetic field on one axis. It requires a `<channel>` tag indicating the field coil to be used, and a new setpoint as described with a `<value>` tag. The change may be abrupt or an optional `<interpolation>` tag can specify a time-varying profile as described in Subsection 3.2.5. The `<value>` tag specifies a new setpoint with a `units` field indicating the number system, and a symbolic expression for the new value. The change, or start of interpolation, begins at the start time of the containing event tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setMagField>
    <channel>"zFieldCoil"</channel>
    <interpolation type="linear">
      <slope>   <literal>"15"</literal>  </slope>
      <offset>  <literal>"15"</literal>  </slope>
    </interpolation>
    <value units="g">
      <literal>"6.0"</literal>
    </value>
  </setMagField>
</event>
...
```

Listing 6.10: setMagField Example

#### 6.2.7.5 setDCElectrode Tag

The `<setDCElectrode>` action defines a change in the DC electrode voltages for the internals of the trap. It requires a `<channel>` tag indicating the electrode to be changed, and a new setpoint as described with a `<value>` tag. The change may be abrupt or an optional `<interpolation>` tag can specify a time-varying profile as described in Subsection 3.2.5. The `<value>` tag specifies a new setpoint with a `units` field indicating the number system, and a symbolic expression for the new value. The change, or start of interpolation, begins at the start time of the containing event tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setDCElectrode>
    <channel>"ElectrodeQ23"</channel>
    <interpolation type="linear">
      <slope>   <literal>"15"</literal>  </slope>
      <offset>  <literal>"15"</literal>  </offset>
    </interpolation>
    <value units="V">
      <literal>"14.77"</literal>
    </value>
  </setDCElectrode>
</event>
...
```

Listing 6.11: setDCElectrode Example

#### 6.2.7.6 setPolarization Tag

The `<setPolarization>` action changes the polarization of one of the laser beams. It requires a `<channel>` tag indicating the beam to be changed, and a new setpoint as described with a `<value>` tag. The change may be abrupt or an optional `<interpolation>` tag can specify a time-varying profile as described in Subsection 3.2.5. The `<value>` tag specifies a new setpoint with a `units` field indicating the number system, and a symbolic expression for the new value. The change, or start of interpolation, begins at the start time of the containing event tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setPolarization>
    <channel>"Raman1"</channel>
    <interpolation type="linear">
      <slope>   <literal>"15"</literal>  </slope>
      <offset>  <literal>"15"</literal>  </offset>
    </interpolation>
    <value>
      <divisionOperator>
        <systemVariable>"pi"</systemVariable>
        <literal>"2"</literal>
      </divisionOperator>
    </value>
  </setPolarization>
</event>
...
```

Listing 6.12: setPolarization Example

#### 6.2.7.7 setDDSFrequency Tag

The `<setDDSFrequency>` action changes the frequency of one of the DDS generator channels. It requires a `<channel>` tag indicating the DDS channel to be changed, and a new setpoint as described with a `<value>` tag. The change may be abrupt or an optional `<interpolation>` tag can specify a time-varying profile as described in Subsection 3.2.5. The `<value>` tag specifies a new setpoint with a `units` field indicating the number system, and a symbolic expression for the new value. The change, or start of interpolation, begins at the start time of the containing event tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setDDSFrequency>
    <channel>"Cooling1"</channel>
    <interpolation type="linear">
      <slope>   <literal>"15"</literal>  </slope>
      <offset>  <literal>"15"</literal>  </offset>
    </interpolation>
    <value units="MHz">
        <literal>"250"</literal>
    </value>
  </setDDSFrequency>
</event>
...
```

Listing 6.13: setDDSFrequency Example

#### 6.2.7.8 setDDSAmplitude Tag

The `<setDDSAmplitude>` action changes the amplitude of one of the DDS generator channels. It requires a `<channel>` tag indicating the DDS channel to be changed, and a new setpoint as described with a `<value>` tag. The change may be abrupt or an optional `<interpolation>` tag can specify a time-varying profile as described in Subsection 3.2.5. The `<value>` tag specifies a new setpoint with a `units` field indicating the number system, and a symbolic expression for the new value. The change, or start of interpolation, begins at the start time of the containing event tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setDDSAmplitude>
    <channel>"Cooling1"</channel>
    <interpolation type="linear">
      <slope>   <literal>"15"</literal>  </slope>
      <offset>  <literal>"15"</literal>  </offset>
    </interpolation>
    <value units="mV">
        <literal>"250"</literal>
    </value>
  </setDDSAmplitude>
</event>
...
```

Listing 6.14: setDDSAmplitude Example

#### 6.2.7.9 setDDSPhase Tag

The `<setDDSFrequency>` action changes the running phase of one of the DDS generator channels. It requires a `<channel>` tag indicating the DDS channel to be changed, and a new setpoint as described with a `<value>` tag. The change may be to set an absolute phase, or a relative phase shift, as indicated by the `type` attribute. The change begins at the start time of the containing event tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setDDSPhase type="relative"/>
    <channel>"Cooling1"</channel>
    <value units="radian">
      <divisionOperator>
        <systemVariable>"pi"</systemVariable>
        <literal>"2"</literal>
      </divisionOperator>
    </value>
  </setDDSPhase>
</event>
...
```

Listing 6.15: setDDSPhase Example

#### 6.2.7.10 ccdMeasurement Tag

The `<ccdMeasurement>` action indicates the start of a new CCD camera measurement. Measurements require a pre-defined resource as described in Subsection 6.2.2. The described resource id is used in the `<resource>` tag to attach the results for later use or download. The exposure time of the camera is controlled by the optional `<integrationTime>` tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <ccdMeasurement>
    <channel>"ionImagingCamera"</channel>
    <resource name="ccdImage1"/>
    <integrationTime units="ms"> <literal>10</literal>  </integrationTime>
  </ccdMeasurement>
</event>
...
```

Listing 6.16: ccdMeasurement Example

#### 6.2.7.11 pmtMeasurement Tag

The `<pmtMeasurement>` action indicates the start of a new PMT counter measurement. Measurements require a pre-defined resource as described in Subsection 6.2.2. The described resource id is used in the `<resource>` tag to attach the results for later use or download. The counting time is controlled by the required `<countTime>` tag. For use with branching logic, the `<decisionThreshold>` tag is required to define the decision rule for state determination[5].

[5]For optical experiments, the collection of photons is a statistical process, and the threshold discriminates true fluorescence from background noise.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <pmtMeasurement>
    <channel>"pmtChannel1"</channel>
    <resource name="pmtMeasurement1"/>
    <countTime units="ms"> <literal>10</literal>  </countTime>
    <decisionThreshold> <literal>"100"</literal> </decisionThreshold>
  </pmtMeasurement>
</event>
...
```

Listing 6.17: pmtMeasurement Example

#### 6.2.7.12 ttlMeasurement Tag

The `<ttlMeasurement>`[6] action allows the measurement of a single TTL input channel's level, or the time in which it changes. It requires specification of the TTL channel using the `<channel>` tag, and a resource for the resulting measurement. The type of transition is recorded over the duration specified in `<duration>` tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <ttlMeasurement>
    <channel>"ttlInput1"</channel>
    <resource name="ttlMeasurement1"/>
    <type>"rising"</type>
    <duration unit="us"> <literal> 15 </literal> </duration>
  </ttlMeasurement>
</event>
...
```

Listing 6.18: ttlMeasurement Example

#### 6.2.7.13 setTTLValue Tag

The `<setTTLValue>` action forces a change to the voltage of a TTL output channel. It requires specification of the output channel using the `<channel>` tag, and the new value

[6] TTL measurement is a placeholder for a generic input operation that may be defined later.

specified by a symbolic expression in the <value> tag.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setTTLValue>
      <channel>"ttlOutput1"</channel>
      <value>
          <literal>"1"</literal>
      </value>
  </setTTLValue>
</event>
...
```

Listing 6.19: setTTLValue Example

#### 6.2.7.14 setPIDcoefs Tag

The <setPIDcoefs> action forces a change to one of the internal feedback stabilization loops, as described in Chapter 7. It requires specification of the output channel using the <channel> tag, and new values specified by a symbolic expression in the <kp>, <ki>, and <kd> tags. PID coefficients are closely related to the overall stability of the trap. As such, only the Calibration program has access to changing these parameters in a quantum program.

```
...
<event>
  <starttime type="relative" units="us">  <literal>"15"</literal> </starttime>
  <setPIDcoefs>
      <channel>"magFieldFeedback-z"</channel>
      <kp>
          <literal>"1"</literal>
      </kp>
      <ki>
          <literal>"0"</literal>
      </ki>
      <kd>
          <literal>"0"</literal>
      </kd>
  </setPIDcoefs>
</event>
...
```

Listing 6.20: setPIDcoefs Example

### 6.2.8 Loop Tags

The majority of the execution flowgraph can be represented in a basic tree shape, and the `<program>`, `<segment>`, `<event>`, and various action tags support this structure. However, loop constructs are difficult to construct in this way, when the loop spans branch points. The *loop* tags support this non-tree shape. Two tags, `<loop-start>` and `<loop-end>`, are designed to indicate the corresponding loop points. Each tag requires an `id` attribute. The `<loop-start>` tag defines the addtional `count` attribute, which also defines the number of loop iterations.

```
<segment>
    <event>
    </event>
    <loop-start id="loop1" count=50/>
    <event>
        <starttime type="relative" unit="us">500</starttime>
        ...
    </event>
    <loop-end id="loop1">
</segment>
```

Listing 6.21: XML Algebra Example

Caution should be exercised using the looping construct. All events within the tag boundaries should be relative time, otherwise the loop may fail to terminate.

## 6.3 Symbolic Algebra Language

The XML intermediate language supports a significant number of arithmetic, geometric and transcendental functions. Examples of these are described in Section 4.6. Algebraic expressions are accessed using the various `operator` tags as listed in Table 6.2.

| Operation | XML Tag |
|---|---|
| $a + b$ | `<sumOperator>` |
| $a - b$ | `<subtractOperator>` |
| $a \div b$ | `<divisionOperator>` |
| $a \times b$ | `<multiplyOperator>` |
| $\sqrt[b]{a}$ | `<rootOperator>` |
| $a^b$ | `<powerOperator>` |
| $e^a$ | `<expOperator>` |
| $\ln(a)$ | `<logOperator>` |
| $\Gamma(a)$ | `<gammaOperator>` |
| $\sin(x)$ | `<sineOperator>` |
| $\cos(x)$ | `<cosineOperator>` |
| $\tan(x)$ | `<tangentOperator>` |
| $\arcsin(x)$ | `<arcsineOperator>` |
| $\arccos(x)$ | `<arccosineOperator>` |
| $\arctan(x)$ | `<arctangentOperator>` |
| $\arctan(y, x)$ | `<arctangent2Operator>` |
| $\sinh(x)$ | `<sinehOperator>` |
| $\cosh(x)$ | `<cosinehOperator>` |
| $\tanh(x)$ | `<tangenthOperator>` |
| $\text{arcsinh}(x)$ | `<arcsinehOperator>` |
| $\text{arccosh}(x)$ | `<arccosinehOperator>` |
| $\text{arctanh}(x)$ | `<arctangenthOperator>` |

Table 6.2: Symbolic Algebra Operators. QuantumIon's user language allows mathematical expressions based on the calibration variables. The standard arithmetic and transcendental functions are supported. Each operation has a corresponding XML tag. (Self-closing and closing tags are omitted)

The typical form of an operator expression is to surround it with the corresponding open and close tags. Operators may require exactly one, exactly two, or an unlimited number of arguments. Algebraic symbols are either the *literal constant*, or a Named Constant. Grouping operators, such as parentheses, are not used since the nested-tree structure of the XML enforces operator precedence explicitly. Two examples of this are shown in Listing 6.22, where the order of operation are interchanged.

```
<!-- solve (1 + x)^2 -->
<powerOperator>
  <sumOperator>
    <literal>"1"</literal>
    <systemVariable>"x"</systemVariable>
  </sumOperator>
  <literal>"2"</literal>
</powerOperator>

...

<!-- solve 1 + x^2 -->
<sumOperator>
  <literal>"1"</literal>
  <powerOperator>
    <systemVariable>"x"</systemVariable>
    <literal>"2"</literal>
  </powerOperator>
</sumOperator>
```

Listing 6.22: XML Algebra Example with Operator Precedence

## 6.4 Resource Allocation

QuantumIon allows users to perform many types of measurements in the course of a program. These measurements are not available for perusal by the user until *after* the program is completed, however the `<decision>` block also makes use of measurements during execution. Clearly the concept of measurement must be something with a meaningful identification.

This problem is solved through the concept of a *resource*. A resource becomes a generic term for all measurements, waveform files, and other data structures that must be coordinated within a program, or outside of it. By pre-defining resources, the QuantumIon server can allocate space, and the user can apply their own identification labels[7].

[7]Such labels must be unique.

## 6.5 Decision Logic on Resources

Resources for raw data collection are treated slightly differently than those used as the basis for branching logic. Both forms are available to the user. While raw data, such as PMT counts or CCD images may be useful in its raw form, branching logic is used to make decisions that change the program execution, and so must be in a binary form to support the type of *if-then-else* lookup table logic required by program change decisions. Raw data images, counts, and values are directly downloadable. They may also be used as the basis for decisions if passed through a decision logic tag which creates a boolean value. Examples of such decision logic might be the threshold of a counter (i.e. is the count below or above a value), or a threshold of a CCD camera (i.e. is an ion brighter or darker than some set value). The values used by such thresholds are adjustable by the user.

## 6.6 Sub-functions

Design re-use is a highly desirable feature of academic research. In the case of QuantumIon, this re-use comes in the form of sub-functions: a pre-defined set of evnets, and associated actions, that are given a name and inserted as needed.

Sub-functions require several basic tags. The `<functionHeader>` tag defines the calling syntax and function parameters. The `<function>` tag defines the function body, with external parameters defined in `<param>` tags. The function is invoked within a segment using the `<useFunction>` tag, where the formal parameters mapped using the `<arg>` tag[8].

The separation between a function definition and its body is important for encrypted programs as described in Section 6.7. Headers are defined in the `<headers>` collection, while function bodies are defined in the `<functions>` collection. Both headers and function bodies must be defined prior to the root segment.

[8]The naming convention comes from typical computer science parlance, where a variable passed to a function is called an *argument*, or *actual parameter*, from the caller's perspective, and a *formal parameter* from the perspective of the function body.

```
<experiment xmlns="https://iqc.uwaterloo.ca/quantumion">
  <headers>
    <functionHeader name="delay-pulse-delay">
      <param>start-delay</param>
      <param>pulse-duration</param>
      <param>end-delay</param>
    </functionHeader>
  </headers>

  <functions>
    <function name="delay-pulse-delay">
      <event>
        <starttime type="relative">start-delay</starttime>
        <simpleLaserPulse>
          <duration>pulse-duration</duration>
          <channel>AOM_CHANNEL_1</channel>
        </simpleLaserPulse>
      </event>
      <event>
        <starttime type="relative">end-delay</starttime>
        <noOp/>
      </event>
    </function>
  </headers>

  <program>
    <root-segment>
        ...
        <useFunction name="delay-pulse-delay">
          <arg name="start-delay">     <literal unit="us">15</literal>  </arg>
          <arg name="pulse-duration">  <literal unit="us">50</literal>  </arg>
          <arg name="end-delay">       <literal unit="us">25</literal>  </arg>
        </usefunction>
        ...
    </root-segment>
  </program>
</experiment>
```

Listing 6.23: XML Subfunction Example

## 6.7 Encrypted Programs

Although QuantumIon is an open, free-access platform for quantum computing experiments, it is recognized that some types of research may use tools that are sensitive, or commercial products, or otherwise not public knowledge. To support this, the sequence

compiler supports *encrypted programs*. This process assumes a third-party developer has created a library of subprograms (i.e. functions, or gates), and provides an encrypted file to be included in the users' programs[9].

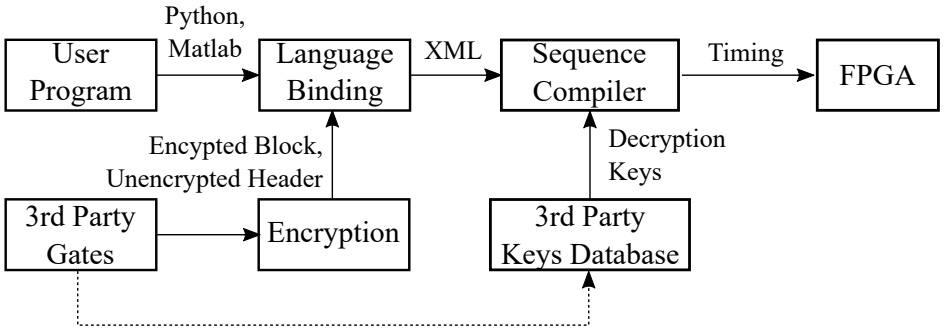


Figure 6.1: Evolution of Encrypted Programs. Encryption uses a third-party developer in addition to the end user. This third party must register a decryption key with the QuantumIon server, and provides their proprietary code as an encrypted XML block to the end user. The QuantumIon server's sequence compiler is the only entity that sees the clear text of these blocks.

Integration of encrypted programs is shown in Figure 6.1. A third-party developer has registered a decryption key with the QuantumIon server ahead of time. When a new set of third-party functions becomes available, the developer encrypts the resulting XML, along with an unencypted *header*, and sends it to the end user. The user writes a quantum program (using one of the language bindings), they make use of these library functions by name. The resulting XML sent to QuantumIon contains both the user XML code and encrypted libraries (and header). When the program is queued for compilation, the sequence compiler (see Section 4.2) decrypts the third-party libraries an proceeds as usual.

Encrypted programs are encoded in the XML as `<secure-library>` and `<secure-header>` tags.

This scheme ensures that only the QuantumIon main program ever actually sees the clear-text library. Further, because the libraries are transmitted using the same XML syntax as the rest of the process, third-party developers use any language binding they are most comfortable with; the end user is not required to use the same language binding.

[9]From this perspective, *user* refers to the end user, not the third-party developer

## 6.8 Python Bindings

Python is one of the most popular programming languages in use today for scientific computing. As such, the language has been given special consideration to ensure the binding of it to the XML intermediate language is efficient and natural.

The primary workhorse is the `quantumion` package, and `qi` object therein. This follows the singleton design pattern, where only one object is instantiated for all use. Once created this object is used to construct the pieces of the quantum program, as well as to communicate with the server.

Basic communication is handled through the methods `qi.login()`, `qi.enqueue()`, `qi.logout()` and `qi.download()`, among others. The purpose of each is self-explanatory.

Resources are created using the `qi.PMTResource()`, `qi.CCDResource()`, etc methods. These objects hold the unique identifiers used during the download of results.

The main program is a sequence of `qi.Event()` and `qi.Decision` blocks, entered as a standard Python list[10]. The actions of each event, such as laser pulses and measurements, are instantiations of various `qi` methods, arranged in a Python list.

A typical example is shown in Listing A.1 of Appendix A. The basic design pattern is as follows:

1. Get a `qi` object using the `import` clause.
2. Establish contact with the QuantumIon server using the `qi.login` method, supplying the certificate credentials for the user's account.
3. Create resource objects for each type of measurement. Such resources can be reused over different quantum programs, but must be unique in any single quantum program.
4. Construct a quantum program's execution graph by providing the `qi.program` object with a Python list of `qi.Event()` objects.
5. Request that the program be queued on the QuantumIon server using the `qi.Enqueue()` method
6. Wait for the program to be completed using the `qi.WaitForCompletion()` method

[10]Python lists preserve the order of the contained objects, and will be used most places where a collection of sub-objects is required.

7. Download results of resources using the `qi.Download()` method.

8. Post-process the resulting data as seen fit.

## 6.9 Matlab Bindings

Matlab is a commercial programming language that is also popular in the scientific community. The core language is built around the concept of arrays of data, and has the usual types of standard data structures: aggregate types (i.e. `struct` and `cell`), as well as associative arrays. The language has a less sophisticated grammar than, Python for example, and it is therefore a good candidate for a standard language binding. Creating a language that supports two so very different programming language paradigms is a challenge, and it is hoped that the effort forces the design team to ensure a consistent usage for any future languages as well.

The Matlab binding is virtually identical to that of Python, except in the major data structures. The same basic flow described above is followed, but the basic object is the Matlab array instead of the Python list. The named-argument syntax[11] that Python supports is not available, so a common syntax of `{'arg1', value..}` is used to emulate this.

An example Matlab program is shown in Listing A.2 of Appendix A.

## 6.10 Conclusion

This chapter described the format of quantum programs in QuantumIon's primary interaction language: XML. XML provides the extensibility needed to support new low-level operations as the project evolves. XML describes the structure of the program in the form of hierarchical *tags*, which enclose all operations that a user can ever control. As such, the list of tags is extensive, and a major body of work on this thesis is to define these in a consistent way.

The XML language allows the the use of the symbolic algebra, one of QuantumIon's powerful features for separating machine parameters from the quantum programs. The other powerful feature of QuantumIon, the ability to change the course of a quantum

[11] Named arguments have the form `function( arg1=..., arg2=... )`.

program based on a measurement, is supported through the use of the execution flowgraph concept. To the best of the author's knowledge, branching logic of this type has never been attempted in a quantum computer for general use before.

XML can be daunting to read, comprehend, and program to the non-specialist. Fortunately, QuantumIon programs are designed for so-called binding to other languages. This allows users to program in comfortable high-level languages, and leverage powerful post-processing tools. Since binding does not change the underlying XML that is generated, QuantumIon programming can be flexible, allowing multiple languages, and even extensions to new ones.

The next chapter discusses a control *not* available to the user: feedback control. Feedback is an internal process that operates at realtime speed, and is used to stabilize the physical processes of the apparatus against changes in time, environment, temperature, and unit-to-unit variations.

# Chapter 7

# Feedback Controllers

In Chapter 3, modules for realtime stabilization were described. These modules continually monitor sensors throughout QuantumIon, and perform continual adjustments to remove time, temperature, and other variations of the equipment itself.

QuantumIon provides a large amount of automation in order to minimize the number of human technicians accessing the apparatus. Limiting the need for human interaction provides a measure of consistency in the machine's operation, as well as providing long-term logging of the daily system environment and operation. Such automation is performed, in part, by a set of feedback controllers which normalize the drifts of system parameters.

Feedback goes hand-in-hand with calibration, described in Chapter 8. The two are distinguished in that calibration is a periodic re-assessment of operation that is performed *between* runs of user quantum programs, whereas feedback controllers run *during* the quantum programs themselves. For the most part, calibration provides mapping of real-world parameters to the associated sensor data (such as intensity-to-voltage for a photodiode and Transimpedance Amplifier (TIA)). Such mappings then become the setpoints that the feedback controllers use for stabilization. Alternatively, feedback and calibration can be compared in terms of the confidence interval of a given parameter: calibration parameters are assumed to be static over the time scale of a given quantum program, but feedback stabilized parameters are expected to so quickly that real-time control is necessary.

## 7.1 Laser Frequency Stabilization

Frequency stabilization on lasers is a direct measurement of the wavelengths being delivered on the beam. For External Cavity Diode Laser (ECDL) lasers, the two main controls are the diode current, and diode temperature. For QuantumIon, an external wavemeter is used as a precision wavelength meter. Such wavemeters provide high precision, but accuracy is provided by the use of an external reference cell (usually a vapor absorption cell). The wavemeter under consideration by QuantumIon requires an external for proprietary data processing. Control of the laser is provided by analog current commands[1]. The current command is derived from a standard PID loop as described in Subsection 3.4.3, except that the input is the wavemeter estimate of the instantaneous frequency, which is provided by to the FPGA via Ethernet.

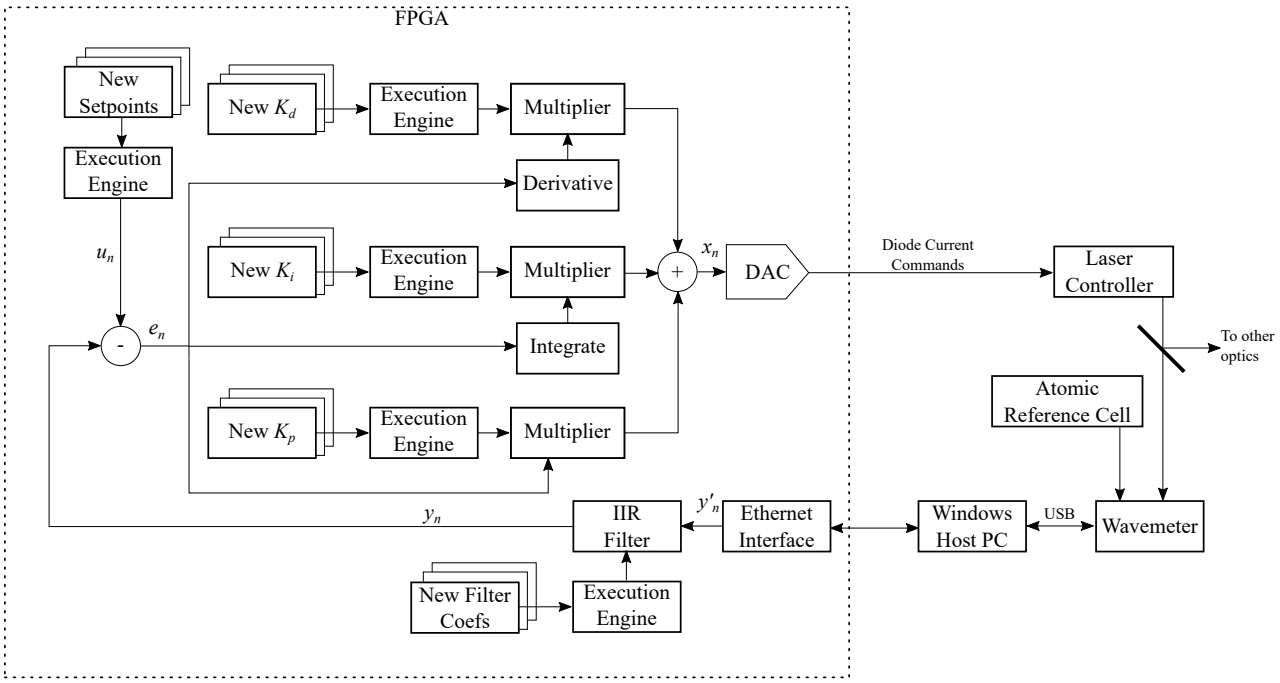


Figure 7.1: Wavelength Stabilization. Stabilization is provided by a standard PID controller, but the feedback source is an external wavemeter. This wavemeter requires an atomic reference and post-processing on a Windows PC. The output of the feedback controller drives the diode current parameter of an external laser controller.

[1]The term *current commands* is used, since the actual laser controller input is a voltage representing the desired current.

## 7.2 Intensity Stabilization

Direct measurement of the intensity of the lasers impinging on the ions is not practical. Further, the optical circuit contains components that either drift with time or temperature, or have unknown calibration parameters. QuantumIon uses various feedback stabilization loops to control these unknowns. The typical stabilization scheme is shown in Figure 7.2.

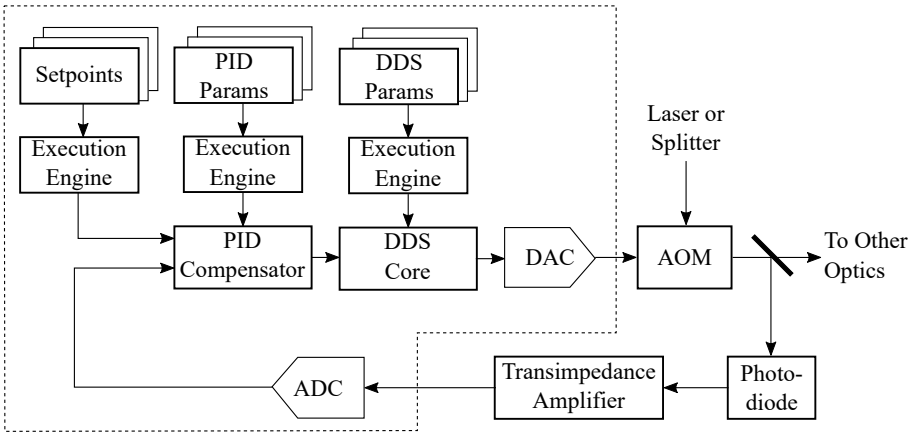


Figure 7.2: Laser Intensity Stabilization. Control of laser intensity is achieved with a standard PID controller and a DDS core that generates an RF frequency. An external acoustooptic modulator alters the intensity of the laser beam in response to changes in RF power. A beamsplitter, photodiode and transimpedance amplifier sample the output beam intensity and complete the feedback loop.

The intensity stabilization loop comprises four major parts: a PID compensator, which controls a DDS frequency generator, an AOM, and a photodiode-transimpedance amplifier pair. The PID provides the overall stabilization and feedback control (see Subsection 3.4.3). The $K_p$ and $K_i$ parameters ensure the long-term stability. The derivative parameter $K_d$ provides improved transient response; this effectively increases the control loop bandwidth which allows a *noise-eater*[2] characteristic. The AOM modulates the incoming laser beam based on an input RF signal. The RF power into the AOM controls the output intensity. Finally, a photodiode is installed downstream of the AOM to provide a sensor. The output of the photodiode is the photocurrent, proportional to intensity. The transimpedance

[2]Noise eater refers to the fact that, although noise is a random phenomenon, if the setpoint is low-variance (akin to a stable reference), a suitably tuned feedback compensator can reduce the variance of the system output. In this case, the variance on laser intensity is reduced.

amp provides a current-to-voltage conversion that is not sensitive to line capacitance; this insensitivity is important for large systems like the QuantumIon, where photodiodes may be some distance from the support electronics.

## 7.3 Raman Beat Note Stabilization

Qubit manipulation as described in Subsection 1.6.3 and Subsection 1.6.4 using two-photon Raman transitions. In order to access the atomic transitions shown in the energy diagram of Figure 1.5, a suitable source of external field is required. For Barium, the transitions are in the microwave X-band of 8-10 GHz. While a conventional microwave RF source has been used extensively and successfully [18][19], the wavelength of these microwave fields is long (tens of centimeters), compared to the atomic spacing (tens of microns). As a result, microwave RF sources cannot address individual ions.

An alternative technique described in [51] replaces the microwave carrier with two counter-propagating optical beams, one detuned such that their difference *beats* at a frequency $\omega_a - \omega_b$. Unfortunately, manipulation of this beat by directly frequency-shifting one beam is difficult with the bandwidth limitations of modern AOM and EOM devices. Instead, practical beats are possible by replacing a pulsed laser source from a mode-locked laser. The electric field provided by the pulse laser is mathematically equivalent to a pure sinusoid that has been amplitude-modulated with a series of repeated sech() functions[3]. In the Fourier domain, this periodic repetition is equivalent to series of Dirac functions with over a broad envelope; this envelope is given by the inverse of the pulse width. The resulting frequency spectrum is a series of frequency replicas that are evenly spaced at integer multiples of the repetition rate $\nu_{rep}$. The desired beat frequency can be formed between the optical carrier, and one of these replicas. The series of spectral replicas appears like a comb shape on a spectrum analyzer, and the $n^{\text{th}}$ spectral replica used to form the beat frequency is often called the $n^{\text{th}}$ comb tooth.

A practical mode-locked laser suffers from environmental factors (temperature, humidity, and acoustic vibration) which can affect the laser cavity mechanical dimensions, resulting a time varying repetition frequency $\nu_{rep}$. The result is a time-varying comb tooth wavelength at

$$\lambda_{tooth}(t) = \frac{c}{c\lambda^{-1} + n\nu_{rep}(t)}. \tag{7.1}$$

Stabilization is described using analog means in [51]. In this technique, the pulse laser

[3]The duty cycle is typically very low, e.g. $t_{on} = 23\text{ps}$ at $nu_{rep} = 2\pi \times 73\text{MHz} \approx 0.17\%$.

passes through an AOM, and the beat frequency is detected with a fast photodiode, which presents the beat envelope (i.e. the $\text{sech}^2$ modulation signal). Since only the $n^{th}$ comb tooth frequency is of interest, the envelope is subsequently down-converted by a precision microwave source to a *baseband* error signal at $\omega_{err}(t) = n \times 2\pi\nu_{rep}(t) - \omega_{ref}$. The microwave source provides the reference for the desired atomic transition frequency.

In QuantumIon, this baseband signal is digitized directly and applied to a PID compensator, which controls the AOM drive frequency. As a result, the AOM drive frequency correspondingly detunes the laser carrier to cancel drifts in the repetition. This technique also has the advantage of amplifying drift in the repetition frequency by the factor $n$, increasing sensitivity as a more distant comb tooth is selected.

When the microwave oscillator is exactly the beat frequency, $\omega_{ref} = n\nu_{rep}$, the error signal is a slow varying signal under most circumstances, i.e. approximately a DC voltage. Unfortunately, output of the photodiode has a large DC component as well, and the two DC voltages become inseparable. Instead, a second ADC channel is used to measure the laser output power, while the first is used to measure the error signal. Due to the precision required in discriminating the error signal, small offset voltages in the ADC converters themselves can cause increased noise on the stabilized beat frequency. To combat this, the microwave oscillator is slightly detuned by an *intermediate frequency* $\omega_{IF}$, and the corresponding ADC signal is demodulated within the FPGA itself[4].

Figure 7.3 shows the major components of the Raman beat note stabilization. In QuantumIon's beatnote stabilization, the core consists of setpoints for AOM power and repetition rate, and the control PID compensator parameters $k_p$, $k_i$, $k_d$. The received signals are digitized by two ADC converters, and are used to estimate the intensity and error. The PID compensator determines the corresponding AOM frequency and AOM amplitude which are fed to the DDS core. An external RF amplifier provides the power needed to drive the AOM. The AOM output, which has been frequency-corrected to stabilize the $n^{th}$ comb tooth, passes through a beam-splitter which samples the laser power sent to the remaining optics. This sampled signal is detected by an ultra-fast photodiode. The photodiode signal is split, with one signal going directly to the ADC, completing the intensity stabilization loop. The other signal ls mixed against the microwave source to produce the error IF signal, which completes the frequency stabilization loop.

[4]The FPGA can down-convert the error signal to DC from the IF frequency with zero error, up to the quantization error of the ADC.

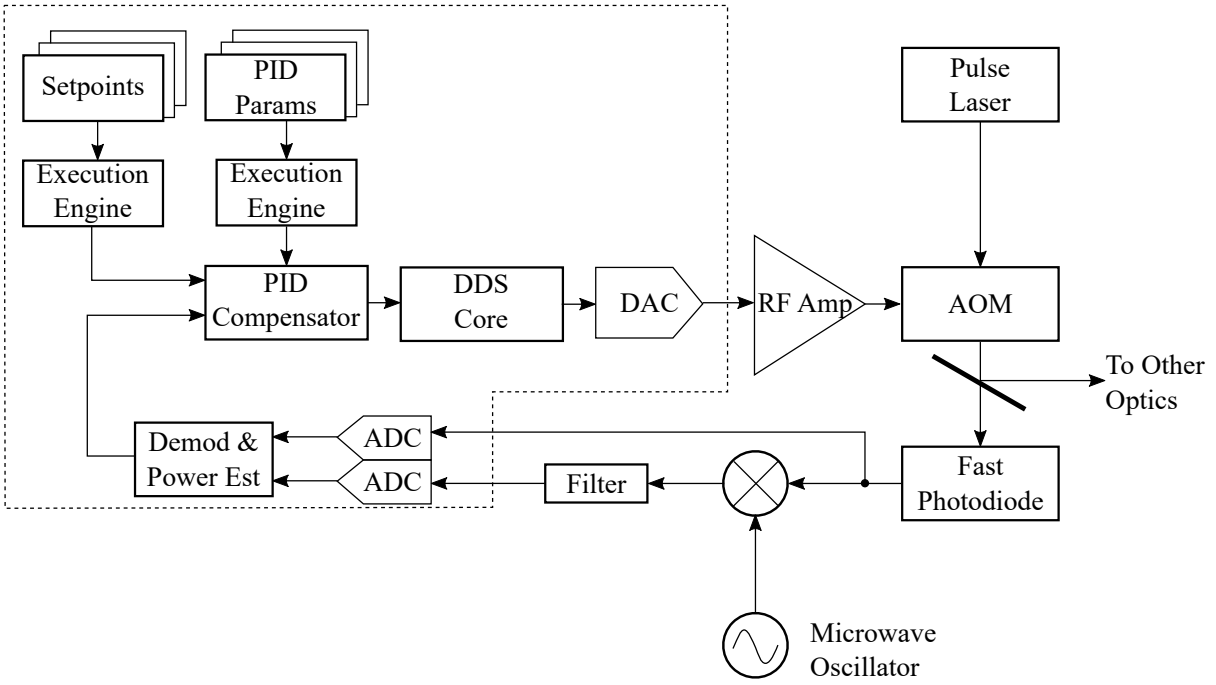


Figure 7.3: Raman Beat Note Stabilization. A PID controller and acoustooptic modulator control the sideband frequency of the pulsed laser beam. The output of a fast photodiode is downconverted via mixing with a precision microwave source. This down conversion selects one of the harmonics of the laser repetition rate, and is used to complete the feedback.

## 7.4 Magnetic Field Stabilization

The magnetic field provides Zeeman splitting to remove degeneracy in the hyperfine states of the ion. The magnetic field is generated by fixed coils mounted on each axis of the vacuum chamber. The coils are excited with a DC current source with a stabilized loop as shown in Figure 7.4.

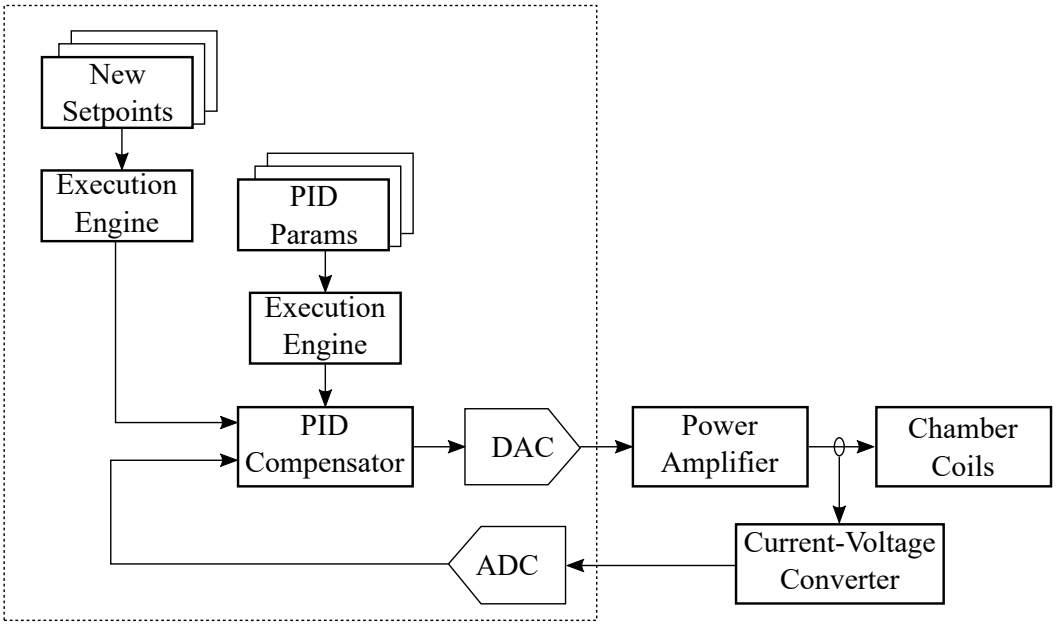


Figure 7.4: Magnetic Field Stabilization. A standard PID controller drives an external power amplifier. The amplifier output drives the chamber coils to provide the hyperfine splitting of Barium. The magnetic field is proportional to the current fed to the coils. Feedback is provided by measuring voltage drop across a small series resistor.

The stabilization loop consists of a PID controller, power amplifier, coils and a current-voltage converter. The PID controller provides the core compensation. The power amplifier provides the high current needed to create the magnetic field. The amplifier has been chosen as a voltage-controlled current source, allowing the user to generate time-varying current during the quantum experiment[5]. Finally, feedback is provided by a current-to-voltage translation stage by measuring the voltage drop across a small series resistor.

## 7.5 RF Amplitude Stabilization

The RF amplitude stabilization module ensures a constant RF power regardless of temperature, gain, and frequency chosen. The system schematic is shown in Figure 7.5.

[5]The coils present a highly inductive load to the amplifier. This presents challenges to the internal amplifier's stability.

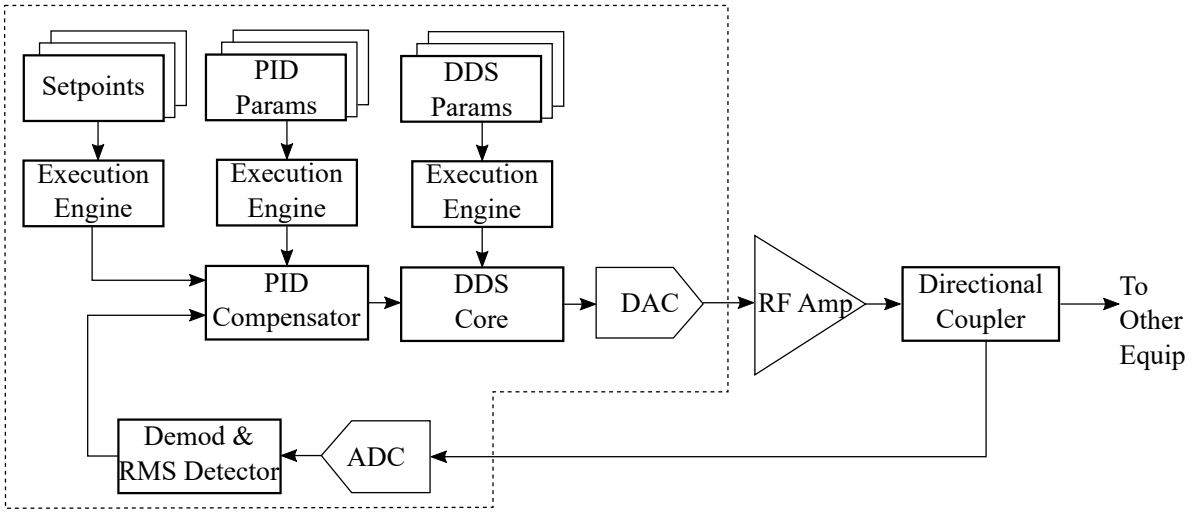


Figure 7.5: RF Amplitude Stabilization. A PID controller drives the amplitude parameters of a DDS core. Drifts in the output amplitude of the amplifier are detected by an RF directional coupler and sent as feedback to the PID. Since the feedback signal is still at RF frequencies, a built-in demodulator is used to extract the amplitude.

The stabilization loop consists of a PID compensate, DDS core, RF amplifier, directional coupler and demodulator, as well as DAC and ADC hardware. The user sets the desired power setpoint. The PID compensator controls the power level sent to the DDS core, which generates the desired frequency as described in Subsection 3.4.4. The RF amplifier is a device with unknown, time-varying gain and so is the primary device to be stabilized. The directional coupler samples the actual amplifier output signal using radiative coupling; in this way the output RF signal is not heavily loaded[6]. The directional coupler signal is at the full RF frequency, and so the demodulator and RMS detector extracts the average amplitude of the generated RF signal and converts it into the linearized measurement. For example, it may be desirable to stabilize in terms of decibels for wide dynamic range, or, alternatively, to stabilize in terms of a linear scale[7].

[6]Typical insertion loss is less than 0.5dB

[7]A decibel linearization does not preserve the noise spectral density, which would grow logarithmically.

## 7.6 Conclusion

This chapter describes the processes used to minimize variations of the physical equipment in QuantumIon. These variations may be caused by ageing, fluctuations with temperature, or simply unit-to-unit differences in individual electronic or optical devices. It is important to understand the difference between feedback and its closely related concept: calibration. Feedback is a realtime process that is not under user control. It is used for those parameters that might change rapidly inside the time of a quantum experiment. Such variations would make QuantumIon unpredictable, unreliable, and not very useful.

The next chapter discusses calibration. It will be shown that through a series of carefully designed quantum programs, results about the details of the QuantumIon apparatus itself are available to the user. A single common database can be read by the user to access these results. The use of these calibration results, when combined with symbolic algebra, provides a very powerful platform for quantum experiments.

# Chapter 8

# System Calibration

In the previous chapter, the concept of feedback and stabilization were introduced, as were the methods used to stabilize QuantumIon's electronics, optics, and mechanical platform against high-frequency variations. In this chapter, the second means, calibration, of decoupling the QuantumIon apparatus from the desired quantum operations will be described.

The use of the symbolic language (see Section 6.3) relies heavily on a set of pre-computed physical calibration values. Doing so relieves the user from incorporating the machine's physical characteristics into his/her quantum program. The *system calibration* is the process wherein these machine characteristics are determined.

The stability of the QuantumIon platform is also indicated by the long-term trends of these calibration parameters. As a result, whenever the calibration database is updated due to a new calibration program, old values are not destroyed, but instead archived with the timestamp of the calibration date. Additionally, a particular calibration value is dependent on the program used to determine it, and so the time-trend of that value is connected to a hash of the source program. In this way, changes to the calibration value can be compared to program changes.

## 8.1 The Role of the Calibrator

The calibration process is performed under a set of credentials for a special user. The user performing calibration is in many respects like any other user. However, a certain very specific set of controls is available to the calibrator that is not otherwise available to the regular user. These controls are restricted because they:

- Can potentially damage the QuantumIon apparatus,
- Could exhaust resources within the sealed trap,
- Can change the state of the machine in non-reversible ways, or
- Can affect the reliability of later results or force a manual recalibration.

As such, there is a need to limit certain critical controls, such as the pulse timing of ablation lasers. As noted previously, there is also a difference between calibration, and stabilization. For example, calibrating the beam power creates a mapping of AOM RF power to delivered beam intensity. Once this mapping is established, the stabilization (feedback) process ensures RF power is kept to the correct value to achieve this intensity. Calibration programs cannot control the feedback loops directly.

There is also the possibility for misuse of the calibration access rights. Calibration is a *role*, not an actual *user*. A particular user assumes calibration rights only while editing calibration programs. After this, they return to standard user rights. As such, the ability to run calibration programs and make changes is only performed from the local network, and at local terminals. It is of critical importance that research programs make use of only standard user rights, in order to preserve the internal consistency of the machine and the user experience.

## 8.2 Calibration Programs

Calibration is performed by a series of automatically-invoked programs. A calibrator edits and develops these programs, and then dictates the schedule at which they must run. Most calibration programs are expected to be time-based, or cycle-based; for example, a calibration might be run every $d$ days (time-based), or after $n$ user programs (cycle-based). In the sections that follow, some of the important and common calibration programs are articulated, along with how such programs fit into the QuantumIon control system as a whole.

Note that this list is neither exhaustive, nor likely to be the best set of calibration techniques; such are the subject of further research. Instead, from a systems engineering standpoint, this list serves to demonstrate the breadth of actions that the QuantumIon control system must perform to provide a useful platform.

### 8.2.1 Calibrating Number of Ions, Position, and Detection of Dark States

Ion count is most directly measured by imaging the ions with the CCD camera. The image processing module described in Subsection 3.4.7 ensures estimates of the size, position, and brightness of each ion, however this module was designed for real-time feedback to support branching logic, and so only simple thresholding and Full-Width Half-Maximum (FWHM) techniques are used. True calibration is likely to require super-resolution techniques (i.e. using multiple images to increase spatial resolution beyond the size of individual pixels). Such techniques require access to the raw image data such as that shown in Figure 8.1.

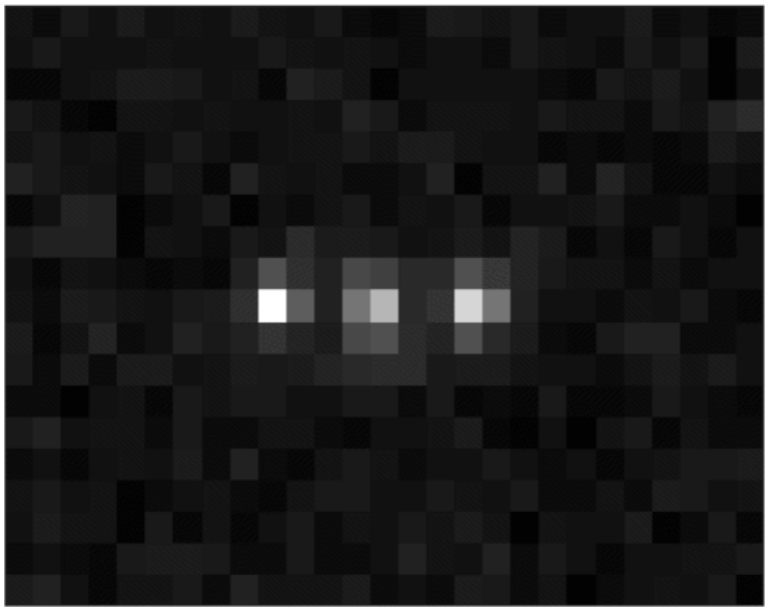

Figure 8.1: Three Barium Ions in a Trap. Raw image data such as this is processed real-time to provide ion position and size. Missing ions indicate a dark state, and positions of ions that are not in a line indicate incorrect trap DC voltages.

One technique for improved estimates is the multiple-hypothesis, model-based technique (see Kay[52], vol I & II). The hypotheses in question are the number of ions in a trap, and the possibility of dark states, where $h_{\text{null}}$ assumes zero ions, $h_b$ assumes one ion in a bright state, $h_d$ assumes a single ion in a dark state, and so on for $h_{bb}$, $h_{bd}$, etc. For each hypothesis, the model-based criterion assumes the CCD data $y_{i,j}$ are weighted combinations of point-spread functions $S(x, y, w)$ for each ion, plus overall noise. The noise is from some

random distribution $\mathcal{N}(\mu, \sigma^2)$ of mean $\mu$ and variance $\sigma^2$. The point spread function of each ion is parameterized by a position $x, y$ and a width parameter $w$. Under the model-based criterion, the calibration involves curve-fitting the received image with that of the aforementioned point-spread functions.

$$h_{\text{null}} = \mathcal{N}(\mu, \sigma^2) \tag{8.1}$$
$$h_{\text{b}} = A_1 S(x_1, y_1, w_1) + \mathcal{N}(\mu, \sigma^2) \tag{8.2}$$
$$h_{\text{d}} = \mathcal{N}(\mu, \sigma^2) \tag{8.3}$$
$$h_{\text{bb}} = A_1 S(x_1, y_1, w_1) + A_2 S(x_2, y_2, w_2) + \mathcal{N}(\mu, \sigma^2) \tag{8.4}$$
$$h_{\text{db}} = A_2 S(x_2, y_2, w_2) + \mathcal{N}(\mu, \sigma^2) \tag{8.5}$$
$$h_{\text{dd}} = \mathcal{N}(\mu, \sigma^2). \tag{8.6}$$

The calibration process involves testing each of these hypotheses against the received data $y_{i,j}$ and choosing the hypothesis whose curve fit gives minimum mean-squared error $(y-h)^2$. The best curve fit yields the parameters $A$, $x$, $y$, and $w$. Examples of point spread functions include the Gaussian function, and the sinc function

$$S_{\text{gauss}}(x, y, w) = \exp\left(-\frac{x^2 + y^2}{2w^2}\right) \tag{8.7}$$
$$S_{\text{sinc}}(x, y, w) = \frac{\sin\left(\pi w \sqrt{x^2 + y^2}\right)}{\pi w \sqrt{x^2 + y^2}}. \tag{8.8}$$

Notice there is ambiguity in several cases: for example, the mathematical descriptions for the null hypotheses (no ions present), and the all-dark hypothesis $h_b$, $h_{bb}$, etc, are identical. There is also ambiguity in permutations of the same combination of bright and dark states during a shift of the trap center, i.e. $h_{dbd} = h_{ddb}$. Such situation can be remedied by improving the point spread functions' position parameters with the expected ion positions; for a given hypothesized number of $n$ ions, the possible values of each $x$ and $y$ are limited[53].

### 8.2.2 Calibrating Rabi Frequency

The Rabi frequency is the rate at which population is transferred between the $|0\rangle$ state and the $|1\rangle$ state. The precise value of this is dependent on the laser intensity, and the ion itself is an excellent sensor. To perform this calibration, the ion is prepared in the

dark state $|0\rangle$, and a resonant pulse of duration $\tau$ is applied to the ion using the Raman beam. The population in the bright state $|1\rangle$ is measured by the detection beam (493nm for Barium). Since the pulse duration can transfer to a superposition of the bright and dark state, measurement collapses to only one of these. Thus a statistic $p_{\mathrm{b}}(\tau)$ over $N$ repetitions can be formed as

$$p_{\mathrm{b}}(\tau) = \frac{n_b}{N}. \tag{8.9}$$

The basic Rabi Flopping operation is a sinusoidal population transfer function. Post processing using a curve-fit technique as shown in [54] can be used to determine the particular frequency.

### 8.2.3 Calibrating Qubit Detection Error

Qubit detection error is the fraction of uncounted states; that is, a qubit is detected erroneously if it is bright but receives no PMT counts, or dark but the PMT records a count anyway. Ideally, the population in dark states plus the population in bright states should reach unity, and there should be zero PMT counts detected during the dark state. For qubits in Barium-133 defined as $|0\rangle = |^6S_{1/2}, F = 0\rangle$ and $|1\rangle = |^6S_{1/2}, F = 1\rangle$.

Of particular importance is the relative PMT count rate between the bright state $|1\rangle$ and dark state $|0\rangle$. To calibrate, the ion is prepared to the $|0\rangle$ state by cooling and optical pumping. The detection beam is tuned to the $|^6P_{1/2}, F = 0\rangle \leftrightarrow |^6S_{1/2}, F = 1\rangle$ transition, which is far enough detuned from the $F = 0$ states to prevent transfer. Clicks detected by the PMT, $n_0$, are recorded over a precise time interval. The state is then prepared to the bright state $|1\rangle$ and, again, PMT counts, $n_1$, are recorded over the same precision time interval. The detector efficiency is the average ratio of the bright counts to total counts

$$\langle \eta_{\mathrm{det}} \rangle = \left\langle \frac{n_1}{n_0 + n_1} \right\rangle. \tag{8.10}$$

The detector dark count $n_0$ is used as a measure of the noise of the detector and associated electronics; such SNR measurements can also be used to predict detection errors. A simplistic model of the detector is that the counts $n_0$ and $n_1$ are respectively, drawn from Poisson distributions with means $\mu_{\mathrm{bright}}$ and $\mu_{\mathrm{dark}}$.

$$n_0 \sim \mathcal{P}(\mu_{\mathrm{dark}}) \tag{8.11}$$

$$n_1 \sim \mathcal{P}(\mu_{\mathrm{bright}}), \tag{8.12}$$

where the Poisson distribution for mean $\mu$ and count $n$ is

$$\mathcal{P}(\mu, n) = \frac{\mu^n e^{-\mu}}{n!}. \tag{8.13}$$

Each PMT sends out a *click* (single TTL pulse) when an incident photon hits the detector. The control electronics must decide how many clicks constitute a bright state, and conversely how few clicks to tolerate before ignoring the count as background noise. For the two possible states (bright and dark), with $k$ clicks counted, there are a total of four permutations of decision:

- Decide $k$ or fewer clicks constitute a dark state, when the true state is indeed dark (this is a correct choice),
- Decide $k$ clicks constitute a bright state, when the true state is indeed bright (this is also a correct choice),
- Decide $k$ clicks constitute a dark state, when the true state is actually bright (this is a *missed* detection),
- Decide $k$ clicks constitute a bright state, when the true state is actually dark (this is a *false alarm*).

The probabilities of such events are labeled $P_{dd}$, $P_{bb}$, $P_m$, and $P_{fa}$ respectively. For error analysis, only the probabilities $P_m$ and $P_{fa}$ are important. Assume there is a hard threshold $K$ that determines if a count registers as bright or dark. As Poisson processes, the Cumulative Distribution Function (CDF), $Q(K)$ is the running sum of the Poisson distribution,

$$P_{dd} = \sum_{n=0}^{K} \mathcal{P}(\mu_{\text{dark}}, n) = Q_{\text{dark}}(K) \tag{8.14}$$

$$P_{bb} = \sum_{n=K+1}^{\infty} \mathcal{P}(\mu_{\text{bright}}, n) = 1 - Q_{\text{bright}}(K) \tag{8.15}$$

$$P_m = \sum_{n=0}^{K} \mathcal{P}(\mu_{\text{bright}}, n) = Q_{\text{bright}}(K) \tag{8.16}$$

$$P_{fa} = \sum_{n=K+1}^{\infty} \mathcal{P}(\mu_{\text{dark}}, n) = 1 - Q_{\text{dark}}(K). \tag{8.17}$$

This result shows that the two dominant error modes, false alarm and missed detection, are dependent by the mean bright and dark counts, and by the threshold $k$. The CDF is monotonically increasing. The missed detection is the left tail of the bright CDF, which increases along with increasing threshold $K$. Similarly the false alarm is the right tail of the dark CDF, which decreases along with increasing threshold. Too low a threshold may decrease missed counts, but increases false alarms. Also in the extreme case, where $K = 0$, the probability of detecting a dark state when there is one is now zero–a dark state is never identified.

The optimum threshold $K$ that minimizes $P_m + P_{fa}$ can be shown to be

$$K_{\text{opt}} = \frac{\mu_{\text{dark}} - \mu_{\text{bright}}}{\log \mu_{\text{dark}} - \log \mu_{\text{bright}}}. \tag{8.18}$$

### 8.2.4 Calibrating Beam Power

Calibrating beam power is a necessary, albeit perhaps one-time, step that is closely related to the stabilization of beam power described in Section 7.2.

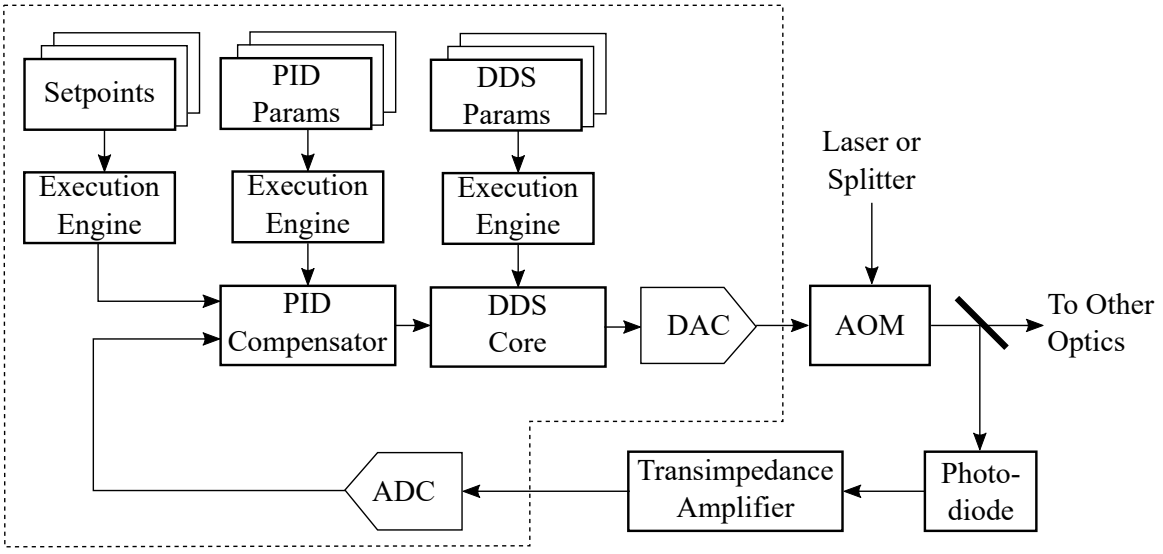


Figure 8.2: Beam Power Calibration reproduced from Section 7.2. The intensity-to-voltage calibration of the combined transimpedance amplifier and photodiode must be mapped as part of system bring-up.

The feedback stabilization routine runs during the course of the program, but the initial mapping of photodiode voltage to intensity must be performed. This is most easily performed with a human operator and a calibrated photodiode for comparison. The human operator makes a measurement of the beam power at a photodiode, and then a measure of the voltage recorded by the ADC. The digital codeword produced by the ADC maps to the measured beam power, and this mapping is the calibrated detector sensitivity. Three points of measurement, of three known intensities, and the corresponding photodiode voltages (from the transimpedance amplifier) ensures the linearity of the measurement.

### 8.2.5 Calibrating Beam Pointing

Optical alignment of each laser beam to its associated lenses and other optical components is often the job of a human technician. In QuantumIon, this task is automated to the extent possible. These alignments fall under the category of *beam pointing*. Typical beam pointing is performed using piezoelectric transducers attached to Thorlabs Polaris series mirror mounts as shown in Figure 8.3. The mirror mount is a three-point contact arrangement, with two orthogonal directions under piezo control. Thus increasing piezo voltage causes deflections in the X-Z and Y-Z planes of the trap.

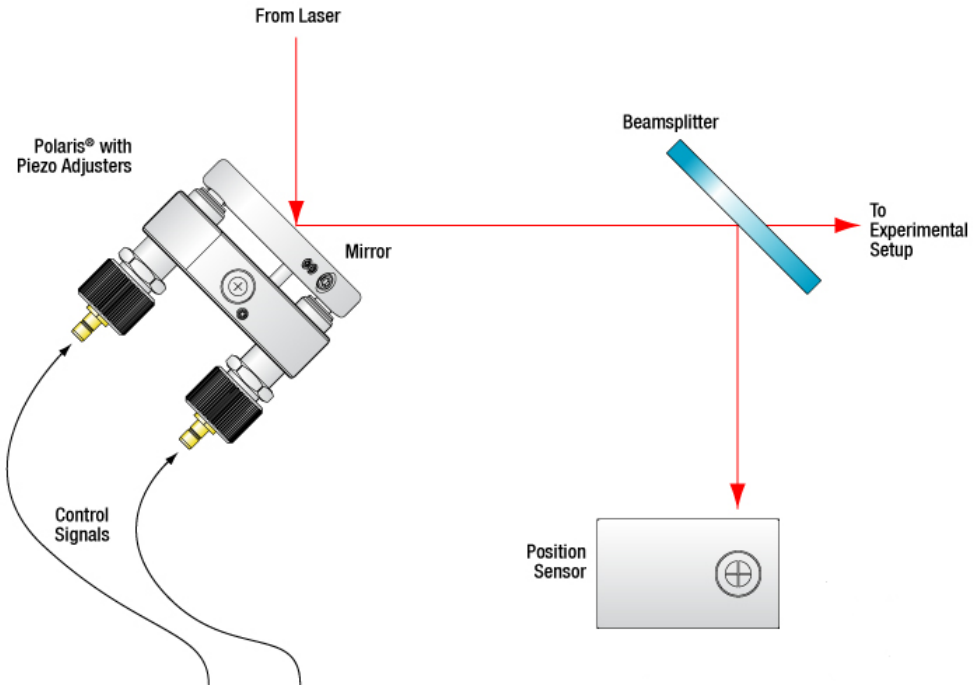


Figure 8.3: Beam Pointing Optics. Laser position is measured by a four-photodiode sensor that measures position. A piezo transducers at the edges of the mirror provide two axes of angular control. When operated in pairs, this type of arrangement can position a beam precisely on a 2D plane, as well as control angle of incidence. Image from https://www.thorlabs.com.

Calibration of the beam is grossly performed with a photodiode position sensor or CCD camera. The ion's Rabi frequency is dependent on the intensity of the received beam, and so the position of mirrors that gives the highest Rabi frequency is the optimum focus for Raman beams. The same can be said for the shelving beam. Cooling, and repump beams do not cause the population transfers that give rise to Rabi frequencies, but the scattering rate is detectable, and the peak scattering is similarly used to find optimum mirror placement[1].

[1] Active feedback pointing is also under consideration, at which point this task becomes a topic of Chapter 7.

### 8.2.6 Calibrating Normal Mode Frequency

The normal mode of the ion chain corresponds to the motional mode the ions experience through Coulomb repulsion, and these modes form the basis of the analysis of the ion trap as a QHO. The energy diagram in Figure 8.4 shows the level structure of one mode of the QHO imposed on the electronic states of the ion.

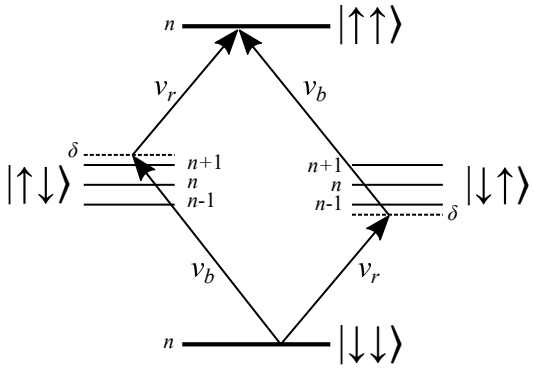


Figure 8.4: Energies for a Single Motional Mode of the MS Gate. The normal mode frequency spectrum is found by sweeping the Raman beam in frequency, and detecting fluorescence.

The energies for a given motional mode with frequency $\omega$ are equally spaced multiples of $\hbar\omega$. As such, transitions between, e.g. the ground electronic state $|\downarrow\downarrow\rangle$ and the $n^{\text{th}}$ motional state cycle, emitting photons in the process. Calibration of these states begins with preparing the electronic state $|\downarrow\downarrow\rangle$ and driving the resonant transition by sweeping the Raman beam AOM. The fluorescence is detected by PMT counts, and the resulting histogram can be analyzed. The result is the sum of regularly-spaced peaks for each of the motional modes (e.g. stretch mode, center-of-mass mode, zig-zag mode, etc). Extracting these modes can be as simple as forming a spectrum using Fast Fourier Transform (FFT) techniques, or more advanced high-resolution techniques[54].

### 8.2.7 Calibrating Micromotion

In an ion trap, the ion has motion within the time-varying electric field driven at frequency $\Omega$. Two motional frequencies experienced in an ideal trap, for each trap axis $x, y, z$. The secular frequency, $\omega_{x,y,z}$ given by the trap parameters, such as ion charge $Q$, mass $m$, and

electric field DC and AC amplitudes $U_0$ and $V_0$, is the primary term used to give the QHO analogy.

$$\omega_{x,y,z} = \frac{1}{2}\Omega\sqrt{a_{x,y,z} + \frac{1}{2}q_{x,y,z}^2}, \tag{8.19}$$

$$a_x = a_y = \frac{1}{2}a_z = -\frac{4Q\kappa U_0}{mZ_0^2\Omega^2} \tag{8.20}$$

$$q_x = q_y = \frac{2QV_0}{mR^2\Omega^2} \quad q_z = 0. \tag{8.21}$$

A second type of motion is at the electric field drive frequency $\Omega$, and is termed micromotion. This amplitude is small, but undesirable as it degrades the QHO approximation. A certain amount of this micromotion is unavoidable, but the amplitude is increased as the ion becomes displaced from the spatial null of the electric field, as well as when the trap RF electrodes are driven out-of-phase with each other. At its extreme, this *excess* micromotion may be visible as a smearing of the ion during fluorescence, but is not well detected at the resolution of typical optics. Instead, detection of excess micromotion can be made by the relative phase between the driven RF field and the detected motion of the ion, or by the ratio of fluorescence rates when on resonance to when detuned by the electric field frequency $\Omega$ [20].

Calibration of the excess micromotion by relative fluorescence rates is easily performed by directly measuring PMT counts $R_0, R_1$ for some time $\tau$, at the atomic frequency $\omega_{\text{laser}} = \omega_{\text{atom}}$ and again at the first electric field sideband frequency $\omega_{\text{laser}} = \omega_{\text{atom}} \pm \Omega$. The ratio $R_1/R_0$ can be used to find to the Doppler shift induced by the excess micromotion.

### 8.2.8 Calibrating Sideband Rabi Frequencies

An ion in a chain responds transitions between ground and excited state $|g\rangle \leftrightarrow |e\rangle$, but also to transitions between motional modes $|n\rangle \leftrightarrow |n+1\rangle$. This coupling is utilized with success in the MS gate gate described in Subsection 1.6.4. Coupling between the combined electronic and motional states $|g\rangle|n\rangle$ and $|e\rangle|n+s\rangle$. The coupling strength is analogous to the Rabi frequency, $\Omega_0$, between electronic states, and is termed *sideband Rabi frequency*, given as [8]

$$\Omega_{n,n+1} = \Omega_0 e^{-\eta^2/2}\eta^s\sqrt{\frac{n!}{(n+s)!}}L_n^s(\eta^2), \tag{8.22}$$

where $\eta$ is the Lamb-Dicke parameter, and $L_n^s(x)$ is the generalized Laguerre polynomial.

An applied electric field on resonance with these sideband frequencies causes a cycling transition between $|g\rangle|n\rangle \leftrightarrow |e\rangle|n+1\rangle$. Sweeping the applied electric field near these resonances and measuring fluorescence intensity can be used to map these sideband Rabi frequencies, by curve-fitting to Equation 8.22.

### 8.2.9 Calibrating Raman Laser Repetition Rate

The beatnoat stabilization described in Section 7.3 ensures ions experience an electric field at the frequency of the qubit atomic transitions. However, this scheme does not directly measure the laser repetition rate itself. Fortunately, the fast photodiode detector used in beatnote stabilization has sufficient bandwidth to measure this frequency directly. Low-resolution methods, such as FFT, are easy to implement. Future work can include more sophisticated data-driven approaches from signal estimation, such as Autoregressive (AR), Autoregressive Moving-Average (ARMA), phase-locked loop, Kalman Filter, and periodigram methods[54].

### 8.2.10 Calibrating Zeeman Shift

The Zeeman shift causes splitting of the energy levels in atomic hyperfine states in the presence of an external magnetic field. In ion traps, an external field is applied through the magnetic field coils located outside the vacuum chamber. Changes in this magnetic field cause shifts in the resonant frequency[2], and so the ion itself becomes a very sensitive magnetometer. The calibration of the Zeeman shift is therefore also a precise measurement of the magnetic field strength at the ion position. The resulting Hamiltonian is

$$\begin{aligned} H_{Zee} &= -\mu \cdot B && (8.23)\\ &= -(g_I I + g_J J)\mu_B \cdot B && (8.24)\\ &= g_F m_F \mu_B \cdot B. && (8.25) \end{aligned}$$

To calibrate the Zeeman shift, the state is prepared to the ${}^6S_{1/2}, F = 0$ state using optical pumping. The computational states $|0\rangle$ and $|1\rangle$ correspond to the $F = 0$ and the

[2]For Barium, this shift is approximately 1.4 MHz/Gauss

entire $F = 1$ sublevels, respectively. Population can be transferred to the $|1\rangle$ state using a Raman transition $\pi$-pulse, detuned far from the $^{133}$Ba 493nm resonance to 532nm. Ordinarily the $F = 1, m_F = 0$ state is used exclusively as $|1\rangle$, since it is insensitive to magnetic fields. However, if the Raman beat is swept in frequency, the Raman transition will transfer population to the $m_F = -1$, $m_F = 0$, and $m_F = +1$ states. The detection beam will not excite any population in the $F = 0$ state, and so applying the detection beam at each sweep point results in a measurement of approximate $|1\rangle$ population versus detuning. The peaks of this curve correspond to each magnetic sublevel $m_F$. These transitions are shown in Figure 8.5.

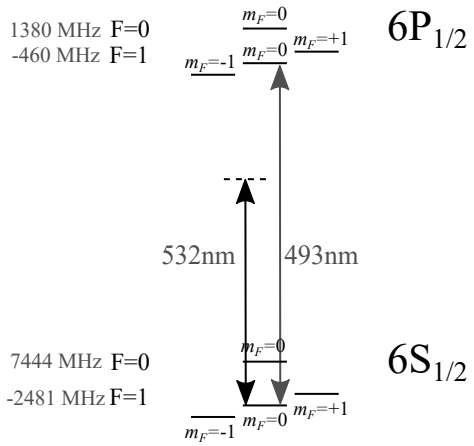


Figure 8.5: Energy Levels for $Ba^+$ Used in Zeeman Calibration

### 8.2.11 Calibrating Laser Intensity Noise

Laser intensity noise manifests itself as intensity fluctuations on the beams themselves. Each beam has its own feedback photodiode that is used for intensity stabilization. To quantify this noise, the variance on this photodiode's voltage can be measured from each intensity compensator. A baseline reading of noise can be performed by recording the raw voltage versus time of the photodiode while the compensator is disabled. The statistical variance $\sigma^2$, and mean $\mu$ are determined from long-term recordings. Later the compensator is re-enabled and the variance and mean are re-computed on the stabilized beam. This gives a figure of merit for the improvement that closed-loop control gives.

### 8.2.12 Calibrating DC Trap Voltages

The basic electronics for the trap DC voltages are described in Subsection 3.4.8. Since no feedback is available within the trap, the electrode voltages are only calibrated once, at the FPGA output. The electrodes themselves are a high-impedance output, and so a negligible leakage current flows through cabling to the trap. The dominant sources of error are expected to be the DAC on the FPGA board itself.

The two most important terms are the offset voltage $V_{\text{ofs}}$ and the voltage gain $G$. The driving voltage $V_{\text{in}}$ comes from the FPGA DAC. Assume a converter with $N$-bit resolution, and with a precision voltage reference $V_{\text{ref}}$. If the FPGA commands the DAC voltage code $n$, these form a linear equation,

$$V_{\text{out}} = V_{\text{ofs}} + G\,V_{\text{in}} \tag{8.26}$$

$$= V_{\text{ofs}} + \left(\frac{G\,V_{\text{ref}}}{2^{N-1}} n\right). \tag{8.27}$$

Equation 8.27 shows that the two primary calibration values, $V_{\text{ofs}}$ and $G$ may be determined by a two-point measurement of the output $V_{out}$ at the two extreme codewords, $n = 0$ and $n = 2^{N-1}$.

Equation 8.27 assumes the gain $G$ in particular is constant across codewords. In reality, DAC converters can suffer from both Integral Non-Linearity (INL) and Differential Non-Linearity (DNL) effects[55]. An improved model is a third-order polynomial of the form

$$V_{\text{out}} = V_{\text{ofs}} + \frac{G_1\,V_{\text{ref}}}{2^{N-1}} n + \left(\frac{G_2\,V_{\text{ref}}}{2^{N-1}}\right)^2 n^2 + \left(\frac{G_3\,V_{\text{ref}}}{2^{N-1}}\right)^3 n^3. \tag{8.28}$$

As before, this calibration can be achieved with a sweep of the codeword $n$, followed by curve fitting the data from a precision voltmeter. This is a human-performed calibration as part of the system bring-up.

### 8.2.13 Calibrating Ion Isotope Population

The process of *loading* the trap with the atomic species begins either by heating a solid sample to the point of vaporization of surface atoms, or by laser ablation. After this photoionization creates a charged atom which is then attracted by the oscillating electric field of the trap electrodes. However, the source material itself, being a high atomic

number (Barium, or Ytterbium are popular choices), may contain many isotopes of the same element. These slight differences in charge-to-mass ratio may still be trapped. The exact isotope of each ion may be determined over the course of an experiment, as each isotope responds to slightly different wavelengths for quantum operations.

Once a chain of ions is trapped, the determination of the isotope begins by preparing the so-called *bright state* of each ion. The QuantumIon optical design, see Section 2.2, provides a single detection beam at 493nm (assuming Barium ions). The fluorescing transition is the $6P_{1/2} \rightarrow 6S_{1/2}$ transition on the energy level diagram. For isotopes with zero nuclear spin, the detunings can be used to determine the isotope. For nonzero nuclear spin, the process is expanded to probe hyperfine states. A detailed spectroscopy of the Barium family is given in [56].

| Isotope | Detuning from $^{138}Ba^{+}$ |
|---|---|
| $^{138}Ba^{+}$ | 0 MHz |
| $^{137}Ba^{+}$ | 271.1 MHz |
| $^{136}Ba^{+}$ | 179.4 MHz |
| $^{135}Ba^{+}$ | 348.6 MHz |
| $^{134}Ba^{+}$ | 222.6 MHz |
| $^{133}Ba^{+}$ | 373 MHz |
| $^{132}Ba^{+}$ | 278.9 MHz |
| $^{130}Ba^{+}$ | 355.3 MHz |

Table 8.1: $6P_{1/2} \rightarrow 6S_{1/2}$ Detunings for Barium Isotopes (from [56])

Table 8.1 above shows the various detunings from the baseline $^{138}Ba^{+}$ line. Odd-numbered isotopes have hyperfine splitting in the GHz range, and so require the larger RF tuning range of EOM modulators, in addition to AOM modulators for fine control. To determine the species, the EOM and AOM drive frequencies of the 493nm detection beam is linearly increased, with CCD images and PMT counts being measured at each point in the sweep. The brightness or darkness of each ion at the detuning frequency in question determines the species. This technique has the advantage of tradeoff between resolution and speed. Slow sweeps allow long integration times and thus low probability of false detection, while faster sweeps improve the turn-around time of the calibration. Note that the branching ratios of Barium are such that long-lived dark states can be populated; therefore during this calibration the optical repump beams must be engaged as well.

### 8.2.14 Calibrating Lab Temperature & Humidity

Measurement of the laboratory environment, especially the temperature and relative humidity, is of importance in diagnosing errors in quantum programs. Changes to temperature and relative humidity can affect laser beam pointing, AOM/EOM efficiency, and camera sensitivity. Calibration of the current state is made by direct measurement of commercial sensors located strategically throughout the lab. Additionally, direct readings are available from the HVAC system. Periodic polling of these parameters not only provides data to be used in symbolic algebra expressions for improving gates (if such information is useful), but to track the trends of the QuantumIon apparatus as a whole.

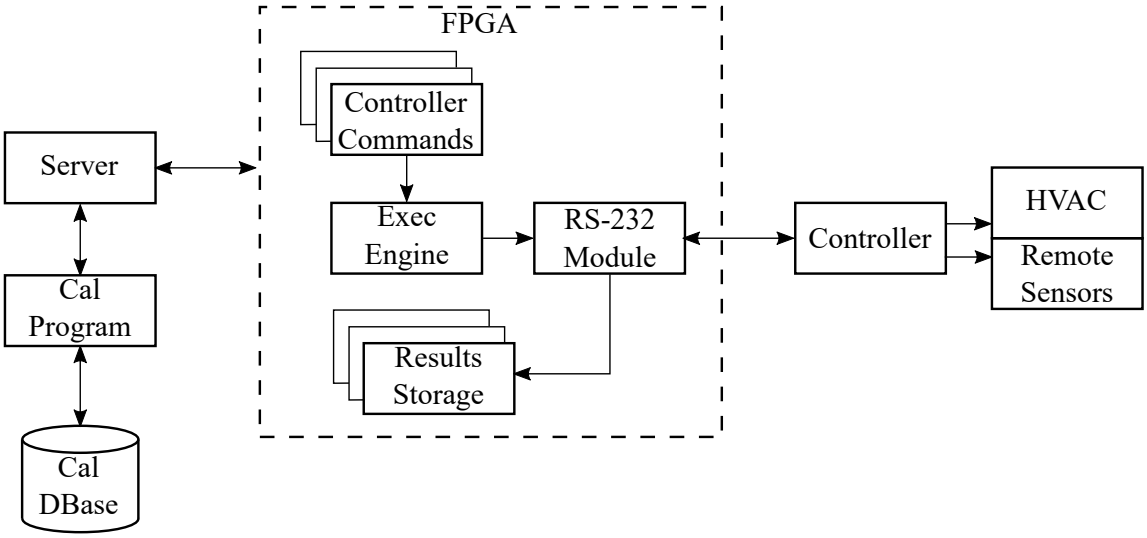


Figure 8.6: Calibration of Lab Temperature and Relative Humidity. These calibration parameters are obtained by direct measurement. Sensors output serial data in the form of RS-232 messages. The execution engine sends request commands and records responses at the same precision timing as all other FPGA functions. Results are stored onto FPGA memory until the end of the measurement program. Data can be post-processed by the calibration program before saving to the database.

### 8.2.15 Calibrating Ion Temperature

Ion temperature in ultra-high vacuum, and in the low-density regime indicates the number $n$ of motional quanta, or phonons, present in the ion chain. The number $n$ is of importance for the Lamb-Dicke approximation needed to treat the ion chain as a simple harmonic

oscillator [19]. The average number of phonons $\bar{n}_\nu$ in a mode $\nu$ can be determined by measuring the absorption of the resolved red and blue sidebands, $S_R$ and $S_B$, of that motional state[57], that is

$$\frac{S_R}{S_B} = \frac{\bar{n}_\nu}{\bar{n}_\nu + 1} = 1 + \frac{1}{\bar{n}_\nu}. \tag{8.29}$$

The measured asymmetry $S_R/S_B$ is easily discernible when near the motional ground state $\bar{n}_\nu \ll 1$, but becomes more difficult to determine as the ion experiences heating. This technique is also sensitive to the intensity calibration of the red and blue sidebands and associated absorption measurement. Another method[58] measures the envelope of long-term Rabi oscillations. Outside the Lamb-Dicke regime, the spin in a single ion does not occur as simple sinusoidal flopping, but as a sum of such sinusoids modulated by the Debye-Waller parameter,

$$\langle n|e^{ikx}|n\rangle = e^{-\eta^2/2} L_n(\eta^2), \tag{8.30}$$

where $L_n$ are the Laguerre polynomials. A curve-fit of the observed population over several Rabi periods can be used to determine the motional population with higher sensitivity.

### 8.2.16 Calibrating Motional Heating Rate

Motional heating rate is the time rate of change in the ion temperature as measured by the number $n$ of motional quanta. The rate can be extrapolated by preparing the ground state, and performing ion temperature measurements as described in Subsection 8.2.15 with suitable wait times. Since no experiments are performed during these temperature measurements, a change in the temperature is indicative of coupling of the ion chain to the thermal bath of the outside environment, heating due to stray fields, or other effects described in [19].

### 8.2.17 Calibrating Vacuum Pressure

Vacuum pressure within the chamber is available through direct measurement of the vacuum gauges inside the chamber. Two such gauges are installed in QuantumIon: a Penning Ionization gauge (cold-cathode gauge), and a Pirani ionization gauge. The cold cathode gauge range is limited to the low-pressure regime, approximately $10^{-3}$ to $10^{-10}$ millibar, while the Pirani gauge operates from atmospheric pressure down to $10^{-3}$ millibar. The

QuantumIon control system communicates with the gauge controller via RS-232 serial commands.

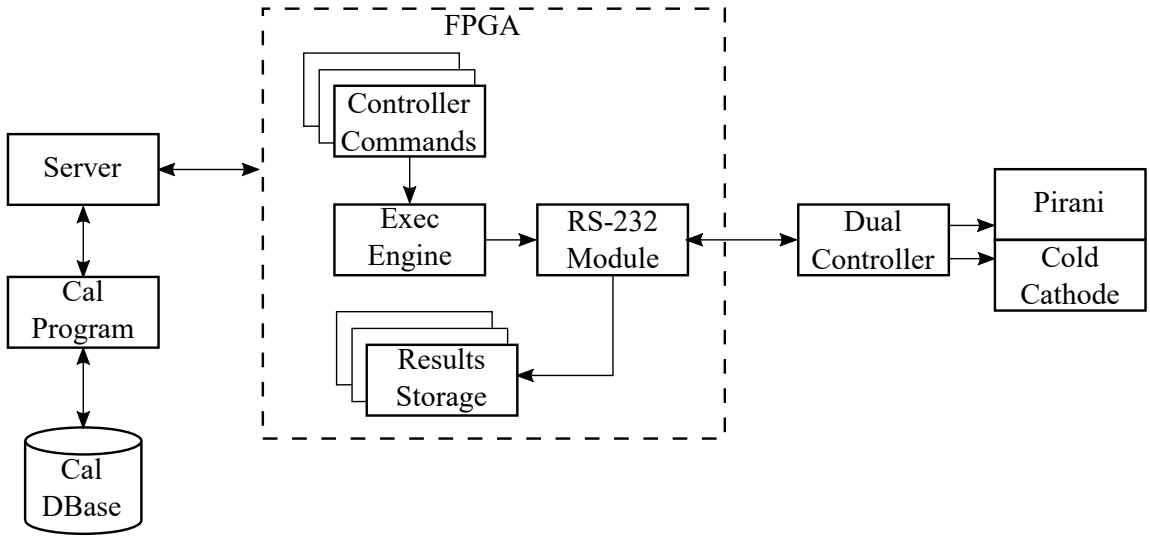


Figure 8.7: Calibration of Vacuum Pressure. These calibration parameters are obtained by direct measurement. The gauge controller outputs serial data in the form of RS-232 messages. Results are stored onto FPGA memory until the end of the measurement program. Data can be post-processed by the calibration program before saving to the database.

### 8.2.18 Calibrating Cooling & Repump Frequency

Cooling and repump lasers are relatively broad in linewidth. To determine the precise cooling and repump atomic transitions, a simple frequency sweep of the AOM driving each beam (493nm and 649nm respectively) can be performed[3]. During cooling, and repumping, the transitions will absorb then re-radiate photons at these wavelengths. The transitions frequencies can be inferred as the AOM detuning that yields peak fluorescence.

### 8.2.19 Calibrating Gate Fidelities

Gate fidelity measurements are a very active area of research. Simple gate tomography and process tomography are common first-pass measurements of fidelity. More sophisticated

[3]A two-dimensional sweep may be difficult to post-process; this procedure can be performed as two independent calibrations.

gate benchmarking, such as randomized benchmarking [59], provide improved results. Under the randomized bechmarking scheme, a random series of unitary gates are applied followed by their own inverses. The final gate is one such that the end state for the permutation is a measurable eigenstate. By performing a statistically significant sequence of these random trials, the statistics of the gates themselves can be determined.

Randomized benchmarking is an excellent example of a fairly low-privilege program. None of the special access that a calibration role provides are required. However, by making this a regularly scheduled measurement of the machine itself, the end user can simply lookup the latest value from the calibration database instead of performing this calibration themself. Additionally, as a regularly scheduled calibration value, the end user or calibrator may benefit from the time history of this measurement; for example, a slow change of fidelity over time may indicate incorrect environmental controls or other outside influences.

### 8.2.20 Calibrating Trap RF Power, Frequency, & Spectrum

The RF power and frequency are generally stabilized using the feedback controllers described in Subsection 3.4.6. However, these feedback controllers require at least one measurement to calibrate the mapping from RF power and frequency to their voltage values. This will also calibrate the tap efficiency of the RF directional coupler indicated in Figure 3.8. This is most easily performed by a sweep of output power and measuring feedback voltage against a calibrated RF power meter.

The output frequency for DDS modules is most sensitive to the accuracy of the Rubidium master clock as described in Subsection 3.2.1. As an atomic standard, the accuracy of the clock source is assumed to be an absolute reference. However, nonlinearities in the DAC output stage, and associated power amplifiers, can create intermodulation products. The combined RF chain, including DAC and amplifier, in terms of a polynomial expansion is

$$v_{out} = a_0 + a_1 v_{in} + a_2 v_{in}^2 + ... \tag{8.31}$$

The DC bias $a_0$ is effectively removed from AC coupled RF circuits, and the linear term is the desired output, where $a_1$ is stabilized by RF power feedback in Section 7.5. The important parameter to calibrate is the second-order term, as this term can create unwanted new frequencies in addition to amplitude errors. The two-tone intermodulation test is effective at determining this. In this test, the DAC generates two non-harmonic

carriers $f_1$ and $f_2$, and a selective filter determines the magnitude at the sum and difference frequencies $f_1+f_2$ and $|f_1-f_2|$. In theory, the polynomial expansion of Equation 8.31 can be expanded to higher-order intermodulation terms. However, modern amplifiers and DACs generally have very small higher-order coefficients, and so measuring such intermodulation *spurs* is likely to require a high dynamic range voltage measurement, such as that from an RF spectrum analyzer.

### 8.2.21 Calibrating Resonator Q Factor and Frequency

The RF resonator Quality factor ($Q$ factor) is a measure of the of the amount of stored energy within the helical resonator cavity. The $Q$ factor is calculated as

$$Q = \frac{1}{R}\sqrt{\frac{L}{C}} = \frac{\omega_0}{\Delta\omega}, \tag{8.32}$$

where $R$, $L$, and $C$ are the resistive loss, and equivalent resonator inductance & capacitance, respectively.

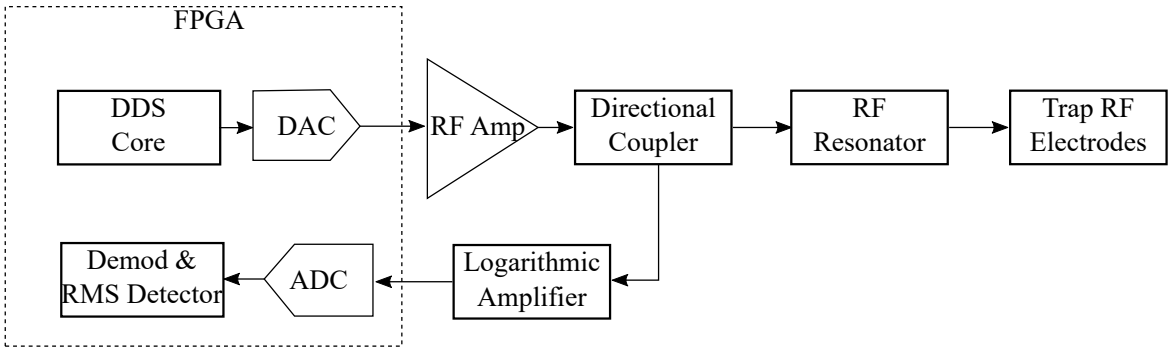


Figure 8.8: Resonator Q Factor Calibration. The FPGA generates a frequency sweep over the aproximate range of the resonator center frequency. The directional coupler measures the reflected power. A logarithmic amplifier improves the dynamic range so the measurement is performed in decibels. The Q factor is the ratio of the peak frequency to the width.

The factors $\Delta\omega$ and $\omega_0$ in Equation 8.32 are the FWHM and the resonant frequency, respectively. To perform the calibration of these parameters, the DDS generator corresponding to the resonator is swept at a fixed rate as shown in Figure 8.8. The return RF

power can be measured as the output of a directional coupler. Since the received signal is an RF waveform, it must be demodulated and converted to power by the RMS detector. The resonant frequency $\omega_0$ is that frequency of lowest return power. Similarly the FWHM $\Delta\omega$ is simply the frequency span centered at $\omega_0$.

It is expected that the resonator $Q$ will change infrequently and this calibration is a general part of the bring-up of QuantumIon. However, a change in $Q$ is an important indicator of a physical change in the overall QuantumIon system. Similarly, a change in return power at $\omega_0$ is an indicator of potential damage to the system. Therefore, periodically re-running this calibration is an important long-term health monitor.

### 8.2.22 Calibrating Detector Dark Counts

The CCD camera, being a physical device, has a finite amount of noise, and knowing this noise floor is important in determining the Signal-to-Noise Ratio (SNR). SNR calcuations are important for determining the optimal threshold for ion detection in the camera image processing (Subsection 3.4.7) and PMT counter module (Subsection 3.4.2).

Dark Counts refer to the electrical noise received by the CCD sensor. To calibrate CCD noise of a given camera, all lasers are turned off, and a series of images is taken from the given sensor. The resulting RMS pixel value is computed with post processing and recorded as a function of integration time.

### 8.2.23 Calibrating EOM Sidebands

Electro-Optic Modulator (EOM) modulators are used to provide precision wavelengths that may be far from that available from commercial tunable diode lasers. Such devices create, for the incoming laser, a region of space with a changing refractive index; this presents a phase-modulated output (see [60], chapter 11) when presented with a sinusoidal driving RF electric field. The efficiency of such a device is nonlinear, obeying a Bessel function mapping of input-to-output intensity. Calibration of this mapping is performed using a Fabry-Perot etalon as an extremely narrow filter whose center is adjustable using a piezoelectric transducer. By sweeping the piezo transducer, an hence the etalon cavity width, the input-to-output intensity law can be mapped from a photodiode at its output as shown in Figure 8.9.

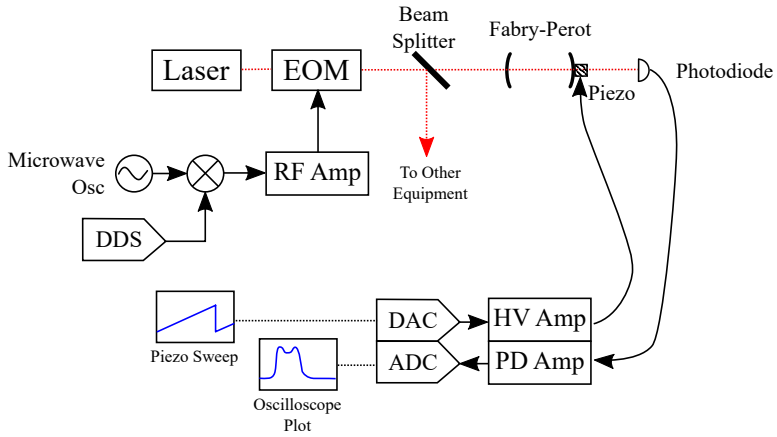


Figure 8.9: EOM Sideband Calibration Setup

### 8.2.24 Calibrating Laser Mode Spectrum

ECDL devices are used throughout the QuantumIon apparatus due to their stability and tunability. Most lasers are of the ECDL variety, except ablation lasers and the Raman lasers. With proper alignment, the ECDL laser supports a single radially-symmetric Gaussian beam, the so-called $TEM_{00}$ mode[61]. However, within the external cavity of the ECDL, improper alignment can support higher-order Gaussian modes which can create undesirable spatial intensities that are not symmetric. The output beam is the sum of each of these modes.

Detection of the modal spectrum is accomplished in two ways: direct imaging of the beam by CCD camera, and use of a scanning Fabry-Perot cavity as a high-precision (i.e. high Q-factor) filter. The CCD camera is capable of resolving spatial modes, but cannot resolve longitudinal modes; such longitudinal modes cause multiple wavelengths to appear at the output[62].

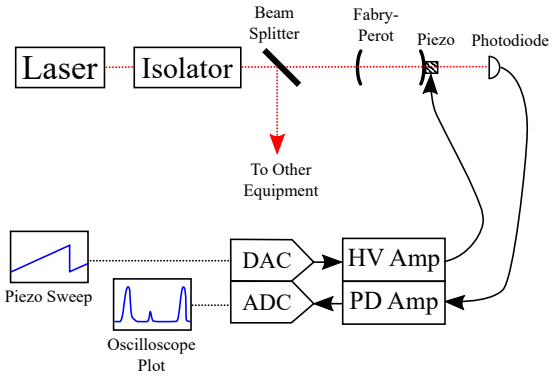


Figure 8.10: Laser Mode Spectrum Calibration

The calibration setup is shown in Figure 8.10. An unmodulated laser is driven into the Fabry-Perot cavity, and the output is detected by a photodiode. The spectrum is determined by sweeping the piezo transducer drive voltage (via a high-voltage amplifier), and graphing the intensity-versus-cavity length curve. Peaks appear at each mode, as well as its multiples over the Free Spectral Range (FSR) of the cavity. The largest peak should be at the $TEM_{00}$ mode, with smaller peaks at the undesirable higher-order modes, and the amplitude of these peaks indicates the amplitude of said modes. Note that such a system can produce backscatter into the ECDL laser cavity, causing loss of lock; hence an isolator is incorporated into the ECDL body.

## 8.3 Conclusion

This chapter, the last one of technical content, described the use of calibration as implemented in QuantumIon. While realtime feedback is used to correct rapid changes in the apparatus, its associated electronics, and optics, calibration is used for slowly-varying phenomenon like drifts due to aging. Additionally, calibration is used for parameters that require a full quantum program to calculate the result; feedback often involves directly-measured variation. The results of a calibration experiment are recorded in the database for use by all. Unlike feedback, the user may choose to disregard the calibrated value in most cases. This allows innovation in many low-level quantum operations. Calibration also introduces the concept of a single privileged set of credentials; calibration programs may allow changes that might otherwise damage the apparatus.

This chapter also lists several example calibration programs that are known at the time of this writing to be critical. The list will expand greatly, and the current list will

change, as the machine is actually built. However, these early examples are useful to explore the full range of the XML programming language and other controls to ensure the QuantumIon control system is indeed capable of the types of quantum experiments needed by researchers.

The next chapter concludes the thesis, and provides a summary of work performed as well as lessons learned.

# Chapter 9

# Conclusion

This thesis describes the control system for the shared-resource quantum computation platform: QuantumIon. Over the course of the preceding chapters, the major functions, equipment, and user interface have been described in detail. Here, the results of this effort are summarized, and the major upgrades and planned improvements are laid out.

## 9.1 Major Features of the Control System

QuantumIon is a platform that should bring the most attractive features of the ion trap quantum computer to a larger number of researchers. To that end, the largest effort in this thesis is in re-designing the classic small, isolated lab paradigm for use on a larger scale. To achieve this goal requires a new direction in thinking about quantum experiments as a whole, and in this spirit the present work is much more than simply a server full of electronics.

### 9.1.1 User-Focused Approach

The QuantumIon control system design began with a comprehensive survey of the different types of users expected to make use of the machine. Over the course of this thesis approximately ten different types of users were identified. Each of these users has a different set of needs. For example, a typical simulation user requires precise, customized control of potentially very long laser modulations. In contrast, an optimization experiment might require a series of single experiments interleaved with sophisticated classical processing

of the measurement results. An ion trap experimentalist working on improvements to the quantum charge-coupled device paradigm requires low-level control of individual beam pointing and ion positions which may vary over time. A quantum algorithms expert might wish to remove themselves from the low-level details of the apparatus altogether, and focus on a gate-level, or circuit level description. And a researcher working on gate performance might wish to tweak the very definitions of these low-level gates, in order to expose physical sensitivities of the apparatus itself. And of course, the researcher working on error correcting codes requires rapid changes to the execution itself on-the-fly. The systems engineering approach taken in this thesis explored each of these conflicting requirements, and the design contained herein is a reflection of attempts to balance all users' needs.

One field of active quantum information research was identified early: Quantum Cryptography and Key Distribution research. This research was not able to find a suitable use for this field in QuantumIon; the hardware and techniques seemed to be specialized on the platform of all-photonic interconnects. However, one of the fortunate aspects of QuantumIon is its flexibility to new programs, and scalability to larger systems, such as hybrid atom-photon interconnects. Researchers have, as an open platform, the opportunity to try new protocols in this field on QuantumIon.

### 9.1.2 FPGA Controls Approach

The sophisticated FPGA based architecture that this design is based on comes from the fact that all research experiments envisioned require pristine control of the time in which changes to the apparatus are made. Such changes take many forms, including changes to AOM and EOM frequencies, RF power, or laser polarization. Similarly, measurements are taken at precise time intervals as well, for example the start/stop times of PMT counters to detect fluorescence must be registered with the other events of the experiment. To this end, the control topology relies heavily on programmable logic.

### 9.1.3 Commercial Off-The-Shelf Hardware

QuantumIon takes the approach of minimal home-built electronics, and instead focuses on systems integration and a homogeneous user experience. This is possible by the use of Commercial Off-The-Shelf (COTS) hardware using standardized connections, interfaces, and physical boards. This decision ensures that equipment has factory support, and replacements and upgrades can be performed long after the original team has left. Further,

the performance demands of the FPGA hardware, and associated inputs and outputs, requires the most advanced computing hardware. In some cases this is the same hardware used in aerospace, medical, and supercomputer data centers. The design of such bleeding-edge technology is well beyond the typical university-level project; it is very much the realm of a specialist company with experience. Additionally, such design, manufacturing, testing, and debugging become distractions to the project. However, the decision to use such technology does come at increased financial cost.

### 9.1.4 Acknowledgement of System Integration Costs

Use of off-the-shelf technology means a software blank-slate in most cases. This is ideal for implementing the total control system described in this thesis; a blank slate has no incompatibilities since it has no capabilities at all. However, it means the software team that implements this control system has to build each module itself. There are several areas where third-party libraries can assist (such as PCI Express interfaces, the TLS security library, symbolic algebras, and interfaces to low-level hardware), but these do not complete, or even come close to completing, the entire system. What is proposed herein is an enormous software undertaking. The benefit of such a design, as is mentioned several times in this thesis, is internal consistency. The user experience, and the calibration experience, and the different levels of user needs, all approach problems in similar ways. As such, there are fewer limitations, surprises, and (it is hoped), a shorter learning curve, in the QuantumIon experience.

### 9.1.5 Scalability and Advanced Networking

Some of the most difficult hardware, and software, involves the high-speed networking required to achieve the generation of arbitrary waveforms. Early on, it was recognized that complete AWG solutions had severe limitations in terms of the ability to play-back very long waveforms; if a product was developed that provided an $x$-millisecond playback, an experiment could be found that needed $2x$ run-time, and upgrade requires entirely new hardware. A drastic new approach from the world of supercomputer designs, which allows the data to essentially stream indefinitely. The only limit now is the size of the storage network–how many hard drives are bought–not the speed of any one link. This approach to scalability can be taken to the extreme in the networking of each FPGA using cutting-edge Infiniband links in addition to the Fibre Channel storage links. This networked approach is also scalable in the number of ions, and the number of traps.

### 9.1.6 Extensive Automation

QuantumIon as a machine makes much more use of automation than previous ion trap designs. In this respect the apparatus is more than just a standard laboratory ion trap with a webserver attached. Over the course of systems engineering, the team was forced to articulate the many different daily tasks that an experimental physicist must perform just to get basic results. Once articulated, these tasks are candidates for automation; after all, the primary goal of QuantumIon is to share access with as many users as possible. No user accesses the hardware as part of their research, but the equivalent result of an experienced graduate student making manual adjustments can be achieved in most cases by using a network of support sensors and computer-controlled motion stages. These sensors ensure alignment of laser beams, stabilization of optical power and frequency, and monitoring of systems. While a human technician is still very much a required team member, the bulk of regular interaction can be performed automatically. This, too, increases cost of the apparatus, but has the benefit of a more reliable platform.

### 9.1.7 New Model for Program Execution

In contrast to other commercial available control systems, and similar built-in-house designs, QuantumIon does not allow the end user to actually write to these FPGA controllers. Therefore a new programming paradigm was invented: the concept of the execution graph. Execution graph analysis is not new in computer science, but the present work attempts to re-evaluate quantum programs from the ground up, starting at this level. In particular, this is, to the author's knowledge, the first attempt to address the concepts of critical-timing events, remote execution, and on-the-fly decision logic together in the same ecosystem. In this respect, QuantumIon's control system is quite unique, and further study in this area could be grounds for exciting and fruitful research.

### 9.1.8 Intermediate User Language

Perhaps the most rewarding, innovative, and important aspect of the QuantumIon control system is the user language. An entire chapter is devoted to it in this thesis, and this reflects the power, modularity, and expressiveness of the different solutions to difficult quantum computing problems. The XML language can be adapted to most any modern high-level language, provided it supports the SOAP interconnection scheme, and has basic data structures such as lists, object types, loops, and text generation. Again, this shows

the belief that QuantumIon is an apparatus that should adapt to changes in scientific programming, including the current standard high-level language.

## 9.2 Future Work

This thesis describes a control system for an apparatus that has not yet been completed. Therefore, the largest area of future work is the implementation of the designs described in the preceding chapters. Every attempt has been made to plan for the most general, long-term solution that meets user requirements. However, there are several areas that are unlikely to be implemented in the first version of QuantumIon.

### 9.2.1 Improved Program Scheduling

The fair scheduling algorithm described in Section 4.3 is very similar to the work done in microprocessor operating system scheduling. It is unknown if the adaptation of this optimization is realistic in the type of workload QuantumIon will enjoy. Undoubtedly, further work in this area will be to the benefit of all, once data on the types and frequency of requests is available.

### 9.2.2 Implementing the Infiniband Network

The Infiniband network described in Subsection 3.2.4 is likely to be deferred until a later version of this hardware and software. The basic reason for this delay is that in a small trap, such as the 16-channel initial version that is planned, the infrastructure costs are quite high, and a simple PCI express network is probably sufficient, and easier to build, test, and integrate. Infiniband will become necessary when multiple traps are joined together.

### 9.2.3 Improved Image Processing

The image processing described in Subsection 3.4.7 is somewhat basic. In particular, it is only expected to estimate ion position, size, and the presence or absence of dark states. This can be a very active area of research in the future.

### 9.2.4 Active Micromotion Compensation

No provision is made in this control system for important improvements such as active, closed-loop micromotion compensation. There have been several proposals for improvements in this area with exciting prospects.

### 9.2.5 Faster Image Capture

Although the high-speed networking employed to achieve nearly infinite playback in the AWG module solves many problems, no time was available to solve similar problems with the capture of ion images from the high-resolution ion camera. As such, there are significant limits to the number and speed of individual ion images that are used for non-realtime purposes (i.e. not used for decision logic). This is a significant hardware and software challenge, and one that deserves a well-thought solution that fits well with the rest of the control system.

## 9.3 Parting Thoughts

A major focus area of the design of this control system was internal consistency and a powerful user experience. Although COTS hardware alleviates the team's need to build circuit boards, the overall software needed to bring the user programming to life is very much the responsibility of QuantumIon's team. This project took the problem of a control system from the ground-up by starting with user requirements, turning those into an idea for how the user language should operate, and matching those to the available, realistic technology. This is in contrast to a more common practice in research of beginning with the available technology, and working back to the resulting user interface. One may call these approaches *requirements-based* and *technology-based* design, respectively.

In taking the former approach, the team must perform all programming and integration itself; there is no single piece of equipment, no single vendor, and no single software library that solves this problem. Meanwhile, the technology-based approach tends to result in a collection of disparate pieces, and is more likely to result in messy, overrun systems integration. It is the author's hope that the this machine, when completed, shows the superiority of the requirements-based approach.

# References


[1] The Cirq Developers. Cirq: A python library for writing, manipulating, and optimizing quantum circuits and running them against quantum computers and simulators. https://github.com/quantumlib/Cirq.

[2] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[3] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[4] Michael A Nielsen and Isaac L Chuang. *Quantum information and quantum computation*. Cambridge: Cambridge University Press, 2 edition, 2010.

[5] Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6):467–488, 1982.

[6] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[7] Lov K Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical review letters*, 79(2):325, 1997.

[8] Dietrich Leibfried, Rainer Blatt, Christopher Monroe, and David Wineland. Quantum dynamics of single trapped ions. *Reviews of Modern Physics*, 75(1):281, 2003.

[9] Dietrich Leibfried, Brian DeMarco, Volker Meyer, David Lucas, Murray Barrett, Joe Britton, Wayne M Itano, B Jelenković, Chris Langer, Till Rosenband, et al. Experimental demonstration of a robust, high-fidelity geometric two ion-qubit phase gate. *Nature*, 422(6930):412–415, 2003.

[10] Guido Pagano, PW Hess, HB Kaplan, WL Tan, Phil Richerme, Patrick Becker, Antonis Kyprianidis, Jiehang Zhang, Eric Birckelbaw, MR Hernandez, et al. Cryogenic trapped-ion system for large scale quantum simulation. *Quantum Science and Technology*, 4(1):014004, 2018.

[11] Shantanu Debnath, Norbert M Linke, Caroline Figgatt, Kevin A Landsman, Kevin Wright, and Christopher Monroe. Demonstration of a small programmable quantum computer with atomic qubits. *Nature*, 536(7614):63, 2016.

[12] Jiehang Zhang, Guido Pagano, Paul W Hess, Antonis Kyprianidis, Patrick Becker, Harvey Kaplan, Alexey V Gorshkov, Z-X Gong, and Christopher Monroe. Observation of a many-body dynamical phase transition with a 53-qubit quantum simulator. *Nature*, 551(7682):601–604, 2017.

[13] Peter Maunz. High optical access trap 2.0. Technical Report SAND2016-0796R, Sandia National Laboratories, January 2016.

[14] Alexander Erhard, Joel J Wallman, Lukas Postler, Michael Meth, Roman Stricker, Esteban A Martinez, Philipp Schindler, Thomas Monz, Joseph Emerson, and Rainer Blatt. Characterizing large-scale quantum computers via cycle benchmarking. *Nature communications*, 10(1):1–7, 2019.

[15] Thomas Monz, Philipp Schindler, Julio T Barreiro, Michael Chwalla, Daniel Nigg, William A Coish, Maximilian Harlander, Wolfgang Hänsel, Markus Hennrich, and Rainer Blatt. 14-qubit entanglement: Creation and coherence. *Physical Review Letters*, 106(13):130506, 2011.

[16] Nicolai Friis, Oliver Marty, Christine Maier, Cornelius Hempel, Milan Holzäpfel, Petar Jurcevic, Martin B Plenio, Marcus Huber, Christian Roos, Rainer Blatt, et al. Observation of entangled states of a fully controlled 20-qubit system. *Physical Review X*, 8(2):021012, 2018.

[17] Steven R. Hirshorn. *NASA Systems Engineering Handbook*. National Aeronautics and Space Administration, Office of Chief Engineer, 2019.

[18] Patricia J Lee. *Quantum information processing with two trapped cadmium ions*. PhD thesis, University of Michigan, 2006.

[19] David J Wineland, C Monroe, Wayne M Itano, Dietrich Leibfried, Brian E King, and Dawn M Meekhof. Experimental issues in coherent quantum-state manipulation of

trapped atomic ions. *Journal of Research of the National Institute of Standards and Technology*, 103(3):259, 1998.

[20] DJ Berkeland, JD Miller, James C Bergquist, Wayne M Itano, and David J Wineland. Minimization of ion micromotion in a paul trap. *Journal of applied physics*, 83(10):5025–5033, 1998.

[21] Steve Olmschenk, Kelly C Younge, David L Moehring, Dzmitry N Matsukevich, Peter Maunz, and Christopher Monroe. Manipulation and detection of a trapped yb+ hyperfine qubit. *Physical Review A*, 76(5):052314, 2007.

[22] Andrew Cox Matt Day Crystal Senko Brendan White, Pei Jiang Low. Practical trapped-ion protocols for universal qudit-based quantum computing. *arXiv*, 2019.

[23] Daniel A Steck. *Quantum and atom optics*, volume 47. http://atomoptics-nas.uoregon.edu/, 2007.

[24] Dmitri Maslov. Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics*, 19(2):023035, 2017.

[25] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. Going beyond bell's theorem, 2007.

[26] Anders Sørensen and Klaus Mølmer. Quantum computation with ions in thermal motion. *Physical review letters*, 82(9):1971, 1999.

[27] PC Haljan, PJ Lee, KA Brickman, M Acton, L Deslauriers, and C Monroe. Entanglement of trapped-ion clock states. *Physical Review A*, 72(6):062316, 2005.

[28] Juan I Cirac and Peter Zoller. Quantum computations with cold trapped ions. *Physical review letters*, 74(20):4091, 1995.

[29] Günther Leschhorn, Taro Hasegawa, and T Schaetz. Efficient photo-ionization for barium ion trapping using a dipole-allowed resonant two-photon transition. *Applied Physics B*, 108(1):159–165, 2012.

[30] PCI Special Interest Group. PCI Express base specification v3.0. Technical report, PCI-SIG, November 2010.

[31] Christopher Monroe and Jungsang Kim. Scaling the ion trap quantum processor. *Science*, 339(6124):1164–1169, 2013.

[32] Alan S Willsky and Nawab S Hamid Oppenheim, Alan V. *Signals and Systems*. Pearson Publishing, 1996.

[33] C Britton Rorabaugh. *DSP Primer with CD-ROM*. McGraw-Hill, Inc., 1998.

[34] Earl E Swartzlander Jr. *Application Specific Processors*, volume 380. Springer Science & Business Media, 2012.

[35] Jack E Volder. The CORDIC trigonometric computing technique. *IRE Transactions on electronic computers*, 1(3):330–334, 1959.

[36] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Internet Engineering Task Force, August 2018.

[37] Anoop Singhal, Theodore Winograd, and Karen Scarfone. Guide to secure web services. Technical Report Special Publication 800-95, National Institute of Standards and Technology, August 2007.

[38] Norman F Ramsey. A molecular beam resonance method with separated oscillating fields. *Physical Review*, 78(6):695, 1950.

[39] Richard B. Kreckel et al Cristian Bauer, Alexander Frink. GiNaC: GiNaC is not a computer algebra system. www.ginac.de/ginac.git.

[40] Martin Gudgin, Anish Karmarkar, Noah Mendelsohn, Yves Lafon, Henrik Frystyk Nielsen, Jean-Jacques Moreau, and Marc Hadley. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[41] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming web services with SOAP: building distributed applications*. " O'Reilly Media, Inc.", 2001.

[42] Taeyoung Choi, Shantanu Debnath, TA Manning, Caroline Figgatt, Z-X Gong, L-M Duan, and Christopher Monroe. Optimal quantum control of multimode couplings between trapped ion qubits for scalable entanglement. *Physical Review Letters*, 112(19):190502, 2014.

[43] Canada National Standard/Canadian Standards ISO/IEC. Information technology - small computer system interface (SCSI) - part 222: Fibre channel protocol for SCSI, second version (FCP-2). Standard, International Organization for Standardization, Geneva, CH, 2016.

[44] T11 Working Group. Fibre channel physical interface-5. Standard, American National Standards Institute, 2010.

[45] PCI-SIG. PCI express base specification version 4.0. Standard, PCI Special Interest Group, 2011.

[46] T10 Working Group. SCSI remote DMA protocol version 2. Standard, American National Standards Institute, 2019.

[47] J. Satran et al. Internet Small Computer Systems Interface (iSCSI). RFC 3720, Internet Engineering Task Force, April 2004.

[48] T11 Working Group. Fibre channel backbone interfaces-5. Standard, American National Standards Institute, 2009.

[49] J. Postel (Ed.). Transmission Control Protocol. RFC 793, Internet Engineering Task Force, September 1981.

[50] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 303–312. IEEE, 2013.

[51] R Islam, WC Campbell, T Choi, SM Clark, CWS Conover, S Debnath, EE Edwards, B Fields, D Hayes, D Hucul, et al. Beat note stabilization of mode-locked lasers for quantum information processing. *Optics letters*, 39(11):3238–3241, 2014.

[52] Steven M Kay. *Fundamentals of statistical signal processing*, volume I & II. Prentice Hall PTR, 1993.

[53] Daniel F.V. James. Quantum dynamics of cold trapped ions with application to quantum computation. *Applied Physics B: Lasers and Optics*, 66(2):181–190, 1998.

[54] Steven M Kay and Stanley Lawrence Marple. Spectrum analysisa modern perspective. *Proceedings of the IEEE*, 69(11):1380–1419, 1981.

[55] Maxim Integrated Circuits. INL/DNL measurements for high-speed analog-to-digital converters (ADCs). Application Note 283, Maxim Integrated Circuits, Inc, November 2001. https://www.maximintegrated.com/en/design/technical-documents/tutorials/2/283.html.

[56] David Hucul, Justin E Christensen, Eric R Hudson, and Wesley C Campbell. Spectroscopy of a synthetic trapped ion qubit. *Physical review letters*, 119(10):100501, 2017.

[57] F Diedrich, JC Bergquist, Wayne M Itano, and DJ Wineland. Laser cooling to the zero-point energy of motion. *Physical Review Letters*, 62(4):403, 1989.

[58] Crystal Senko. *Dynamics and Excited States of Quantum Many-body Spin Systems with Trapped Ions*. PhD thesis, University of Maryland, 2014.

[59] Emanuel Knill, Dietrich Leibfried, Rolf Reichle, Joe Britton, R Brad Blakestad, John D Jost, Chris Langer, Roee Ozeri, Signe Seidelin, and David J Wineland. Randomized benchmarking of quantum gates. *Physical Review A*, 77(1):012307, 2008.

[60] Robert W Boyd. *Nonlinear optics*. Elsevier, 2003.

[61] Joseph Thomas Verdeyen. *Laser Electronics*. Prentice Hall Englewood Cliffs, NJ, 2nd edition, 1989.

[62] Dana Z Anderson. Alignment of resonant optical cavities. *Applied Optics*, 23(17):2944–2949, 1984.

[63] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[64] Jayaram Bhasker. *A VHDL Primer*. Prentice-Hall, 1999.

[65] Jayaram Bhasker. *A Verilog HDL Primer*. Star Galaxy Publishing, 1999.

[66] C. M. Sperberg-McQueen Eve Maler Franois Yergeau John Cowan Tim Bray, Jean Paoli. Extensible markup language 1.0. Technical report, World Wide Web Consortium, 2008.

[67] Jaehyeong Kim and K Konstantinou. Digital predistortion of wideband signals based on power amplifier model with memory. *Electronics Letters*, 37(23):1417–1418, 2001.

[68] Akis Goutzoulis, Dennis Pape, and Sergei (eds) Kulakov. *Design and fabrication of acousto-optic devices*. Marcel Dekkar, 1994.

[69] Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. Full-stack, real-system quantum computer studies: Architectural comparisons and design insights. *arXiv preprint arXiv:1905.11349*, 2019.

[70] Paul C Haljan, K-A Brickman, Louis Deslauriers, Patricia J Lee, and Christopher Monroe. Spin-dependent forces on trapped ions for phase-stable quantum gates and entangled states of spin and motion. *Physical review letters*, 94(15):153602, 2005.

[71] H Häffner, S Gulde, M Riebe, G Lancaster, C Becher, J Eschner, F Schmidt-Kaler, and Rainer Blatt. Precision measurement and compensation of optical stark shifts for an ion-trap quantum processor. *Physical review letters*, 90(14):143602, 2003.

[72] M Teach, Matsuo Kunaiki, and B Salaeh. Excess noise factors for conventional and superlattice avalanche photodiodes and photomultiplier tubes. *IEEE Journal of Quantum Electronics*, (8):1184–1193, 1986.

# APPENDICES

# Appendix A

# Code Examples

The sections below illustrate complete code blocks in the various languages used througout this thesis.

## A.1 Example Python Program

The Python language binding is described in Section 6.8. Below is a basic program for modulating one laser pulse and returning a range of statistics.

```
"""
This program runs a single n=0..100 iteration of a cooling pulse and collects
PMT counts.  The user generates this program and later downloads the results
"""

# need to read files from hard disk
import os

# use matplotlib to draw plots of data
import matplotlib as plt

# get the quantumion master object
import quantumion as qi


# get login credentials (probably this is an SSL certificate)
credentials = open('mycertificate.crt')

# log into the server.  this also creates a session for the "qi" object
qi.login(credentials)
```

```
# create a list for all results
allResults = []

# queue up 100 experiments
for n=range(1,100):
    # create a resource representing the results of a PMT counter for saving later
    pmtResults1 = qi.PMTResource()

    # create the main quantum program as a list of Event() objects
    qi.program = [

        # state preparation and initialization
        qi.StatePrepStep(),

        # Run the cooling laser for "n" microseconds
        qi.Event(
            time=qi.microsecond(500),
            action=[
                qi.SimpleLaserPulse(
                    duration=qi.microsecond(n),
                    channel=qi.CoolingLaser1
                )
            ]
        ),

        # run the detction and pmt counts simultaneously
        qi.Event(
            time=qi.microsecond(700),
            action=[
                qi.SimpleLaserPulse(
                    channel=qi.DetectionLaser1,
                    duration=qi.microseconds(10)
                ),
                qi.PMTMeasurement(
                    resource=pmtResults1,
                    channel=qi.PMTChannel1,
                    counttime=qi.microsecond(10)
                )
            ]
        )

    ]
    # end of main program

    # try to enqueue the program, or die with a message
    try:
        # load the program into the QI server
        qi.Enqueue()
        qi.WaitForCompletion( maxtime=qi.minutes(10) )

        # successful completed experiment.  get the results from this
        # predsefined resource
        pmtCounter = qi.Download(pmtResults1)
```

```
        # the returned download object has lots of other data...just save the counts
        allResults[n] = pmtCounter.CounterValue
    except:
        qi.GenericException,  print("An error ocurred: code is " + qi.GetLastException()+"\n")


# done with experimental data...post process
plt.plot( *zip(*sorted(allResults.items())) )
plt.show()

# shutdown
qi.logout()
```

Listing A.1: Example program in the Python language binding

## A.2 Example Matlab Program

The Matlab language binding is described in Section 6.9. Below is a basic program for modulating one laser pulse and returning a range of statistics.

```
%
% This program runs a single n=0..100 iteration of a cooling pulse and collects
% PMT counts.  The user generates this program and later downloads the results
%

% open the quantumion library
import quantumion;
qi = quantumion;

% get login credentials as an SSL certificate
credentials = fileread('mycertificate.crt');

% log into server...this also creates the login session
qi.login(credentials);

% create an array of structs for all results
allResults = [];

% queue up 100 experiments
for n=(0:100)
  % create a resource representing a PMT counter for saving later
  pmtResults1 = quantumion.PMTResource;

  % create the main program as a celll array of Event
  qi.program = { ...

        % state preparation and initialization
        qi.StatePrepStep(),
```

```
        % Run the cooling laser for "n" microseconds
        qi.Event(
              { 'time', qi.microsecond(500) },
              { 'action', {
                  qi.SimpleLaserPulse(
                    {'duration', qi.microsecond(n) },
                    {'channel', qi.CoolingLaser1 }
                  )
              } }
        ),

        % run the detction and pmt counts simultaneously
        qi.Event(
            { 'time', qi.microsecond(700) },
            { 'action', {
                  qi.SimpleLaserPulse(
                    { 'channel', qi.DetectionLaser1 },
                    { 'duration', qi.microseconds(10) }
                  ),
                qi.PMTMeasurement(
                    {  'resource', pmtResults1  },
                    {  'channel', qi.PMTChannel1  },
                    {  'counttime', qi.microsecond(10) }
                )
            } }
        )
    };
    % end of main program

    % try to enqueue the program
    try
      % send the program to the main QI server, with a timeout of 10 minutes
      qi.Enqueue();
      qi.WaitForCompletion( {'maxtime', qi.minutes(10) } );

      % successful completion
      pmtCounter = qi.Download( pmtResults1 );

      % the returned download object has lots of other data...just save the counts
      allResults = [ allResults, pmtCounter.CounterValue ];
    catch ME
      if (strcmp(ME.identifier, 'QuantumIon:GeneralException'))
        disp( ['An error occured: ', qi.GetLastException() ] );
      end
    end

end % end for loop


% done with experimental data...post-process
plot( allResults );

% shutdown
qi.logout()
```

Listing A.2: Example quantum program using Matlab language binding

## A.3 Example XML Program

```
<!-- XML realization for 2-pulse, awg and ccd program -->
<experiment>
  <resources>
    <file type="awgsource" name="mine.mat">
      <id>12345_created_by_qi_Alloc_function</id>
    </file>
    <file type="ccdraw">
      <id>34567_created_by_qi_Alloc_function</id>
    </file>
  </resources>
  <segments>
    <segment>
      <predefinedstep name="DefaultTrapSetup" />
      <predefinedstep name="DefaultStatePrepStep">
        <arg>"0000"</arg>
      </predefinedstep>
      <step>
        <simplelaserpulse>
          <starttime type="absolute" unit="us">0.2</starttime>
          <duration type="absolute" unit="us">0.3</duration>
        </simplelaserpulse>

        <simplelaserpulse>
          <starttime type="absolute" unit="us">0.8</starttime>
          <duration type="absolute" unit="us">0.9</duration>
        </simplelaserpulse>

        <AWGlaserpulse>
          <starttime type="absolute" unit="us">1.0</starttime>
          <source>
            <id>12345_created_by_qi_Alloc_function</id>
          </source>
        </AWGlaserpulse>

        <CCDImageMeasure>
          <starttime type="absolute" unit="us">1.1</starttime>
          <destination>
            <id>34567_created_by_qi_Alloc_function</id>
          </destination>
        </CCDImageMeasure>
      </step>
    <segment>
  </segments>
</experiment>
```

Listing A.3: Example XML Program

## A.4 Example VHDL Program

This example shows the use of both Behavioral Logic (in the *process* sections) and Combinatorial Logic (at the bottom with the <= assignments).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity FIFO_Comparator is
  Generic (
    VALUE_WIDTH : integer;
    COUNT_WIDTH : integer
  );
  Port (
    reset :             in std_logic ;    -- active-high reset signal,
                                          --   sync with experiment clock
    exp_clk :           in std_logic ;    -- experiment clock
    start :             in std_logic ;    -- active high start trigger
    stop :              in std_logic ;    -- active-high stop sequence

    -- counter FIFO
    count_fifo_do:      in std_logic_vector(COUNT_WIDTH-1 downto 0);
    count_fifo_rden:    out std_logic ;
    count_fifo_empty:   in std_logic ;

    -- value FIFO
    value_fifo_do:      in std_logic_vector(VALUE_WIDTH-1 downto 0);
    value_fifo_rden:   out std_logic ;
    value_fifo_empty:   in std_logic ;

    -- Output Pins
    running:            out std_logic ;
    err_len_mismatch:   out std_logic ;
    err_sequence:       out std_logic ;
    done:               out std_logic ;
    pinout:             out std_logic_vector (VALUE_WIDTH-1 downto 0)
  );
end FIFO_Comparator;
```

```
-- Behavioral model of the component
architecture Behavioral of FIFO_Comparator is
  signal pins_out : std_logic_vector (VALUE_WIDTH-1 downto 0);
  signal count_out : std_logic_vector (COUNT_WIDTH-1 downto 0);
  signal running_out: std_logic;
  signal rden_out : std_logic;
  signal done_out : std_logic;
  signal seq_err_out : std_logic;
  signal mismatch_err_out : std_logic;
begin

  -- simple counter logic with reset
  pCOUNTER: process(exp_clk)
    variable count : unsigned(COUNT_WIDTH-1 downto 0);
  begin
    if(rising_edge(exp_clk)) then
      if( reset = '1' ) then
        count := TO_UNSIGNED(0, count'length);
      elsif( running = '1' ) then
        count := count + 1;
      end if;
    end if;

    count_out <= std_logic_vector(count);
  end process pCOUNTER;



  -- triggering and termination
  pRUNNING: process(exp_clk)
    variable is_running : std_logic ;
    variable is_done : std_logic;
    variable is_seq_err : std_logic;
    variable is_mismatch_err : std_logic;
  begin
    if(rising_edge(exp_clk)) then
      if( reset = '1' ) then
        is_running := '0';
        is_done := '0';
        is_seq_err := '0';
        is_mismatch_err := '0';
      elsif( (is_running = '1') and
          ((count_fifo_empty xor value_fifo_empty) = '1') ) then
        is_running := '0';
        is_mismatch_err := '1';
      --elsif( (is_running = '1') and
          (unsigned(count_out) > unsigned(count_fifo_do)) ) then
      --  is_running := '0';
      --  is_seq_err := '1';
      elsif( (is_running = '1') and
          (count_fifo_empty = '1' or value_fifo_empty = '1') ) then
        is_running := '0';
        is_done := '1';
      elsif( start = '1' ) then
```

```
        is_running := '1';
      end if;
    end if;

    running_out <= is_running;
    done_out <= is_done;
    seq_err_out <= is_seq_err;
    mismatch_err_out <= is_mismatch_err;
  end process pRUNNING;


  -- integer compare and consume FIFO
  pCOMPARE: process(exp_clk)
    variable rden : std_logic ;
    variable pins : std_logic_vector (VALUE_WIDTH-1 downto 0);
  begin
    if(rising_edge(exp_clk)) then
      if(reset = '1') then
        rden := '0';
        pins := (others => '0');
      elsif( running_out = '1' ) then
        if( count_out = count_fifo_do ) then
          pins := value_fifo_do;
          rden := '1';
        else
          rden := '0';
        end if;
      end if;
    end if;

    rden_out <= rden;
    pins_out <= pins;
  end process pCOMPARE;


  -- output mapping
  err_len_mismatch <= mismatch_err_out;
  err_sequence <= seq_err_out;
  value_fifo_rden <= rden_out;
  count_fifo_rden <= rden_out;
  pinout <= pins_out;
  running <= running_out;
  done <= done_out;
end Behavioral;
```

Listing A.4: Example VHDL Program

# Appendix B

# Generating Encrypted Gate Sets

Encrypted gate sets are described in Section 6.7. Here provides the operations necessary to create and transmit functions in a secure manner.

## B.1 Basic Operations

```
<experiment xmlns="https://iqc.uwaterlool.ca/quantumion">
  <program>
    <root-segment>
        ...
        <!-- some measurement corresponding to measurement-1 -->
        <decision resource="measurement-1">
          <condition state="xxxx_xxxx_xxxx_xxx0">
            ...
          </condition>
          <condition state="xxxx_xxxx_xxxx_xxx1">
            ...
          </condition>
        </decision>
    </root-segment>
  </program>
</experiment>
```

Listing B.1: Encryption Example Code

The third-party developer creates an encryption key file and sends it to the QuantumIon server. The `base64` encoding ensures that all characters are printable and thus can be encoded in XML format.

```
openssl rand -base64 > key.bin
```

The result is

```
S/zpO96MIj0BjDKjQZq2xyY16amEK5NqkaTy7PGwgHU=
```

The end user must have a means to identify the key used for decryption. Encryption of gates uses a symmetric key cipher: AES-256. As such, sending the key itself compromises the security of the encrypted gates. Instead, the key is identified using its SHA-256 fingerprint.

```
cat key.bin | openssl dgst -sha256
```

The result is

```
a8add4003bf9e1eed2aca713db9bacc0d453625f001e3d5353ac748df920c994
```

The resulting base-64 data is clearly encrypted.

```
U2FsdGVkX18qw/UtRX41CRVqKclulB2FrmVjBhf1MLc/rmUazjTHk34wPG2yRDUl
MB+kcQYjGAl1GiWNtZRh5kcSHTFtFglDeBpAcBltRWYTWGwYehR2frmqKfYBYaN9
dQdsPxF0PcnP6EuyCpXU/2sm8WYCKF4TVftNRI2Xd1uC5ZctVBfeFlPNw+hSxt/I
DXjoC115nprvvwj/ouQUiOVgwiO1BLMvFx7fURfyJwjOHbO8dVhSLWTVFYD/kD2i
W5/PFoIvkxhB1CxIcB8y4cYzCs7rGJJomI1DLxfYUGES9eWw0u+2jZStcPuDlqQ7
rU2fUEkNzuqG2ZOzT0ncvRA5b7RBm4gU0Run0xNBFy+7BPv0kAn9nRfCCPYACr2g
gJZecxx3WyKgiYbe4jT4P1TUvicK3aNv3h6H9Zr/Nn74uA3DIpk4BVGmOxAY4RZX
VU1gGgsjJ/PaylLqdWw1UXZBkDk+E/NEayLY7tVauLUEIOdqxf5CtHyHrFT5So6g
E1yfURIme+opXGHziffw0aDPDWcxrrTtfINK+fzROTKh7PIM/sm6OWoi3PgXRB/X
tkYJe1TOu71rZBE5Xj/9ghssVTgG9dSY73O6vTWmLs0=
```

For diagnostic, the data can be decrypted as follows:

```
openssl enc -d -aes-256-cbc -base64 -in my.xml.enc -out my.xml.plain
  -pass file:./key.bin
```

## B.2 Encoding Formats

The encryption of third-party gates uses the symmetric-key AES-256 cipher. As a symmetric key cipher, the encryption/decryption key must reside on the QuantumIon server. All transfers to the QuantumIon server are in the form of TLS encrypted transfers. This scheme ensures security over the Internet, and as such the key can be encoded as plaintext XML. The key transfer is between the third-party developer and the QuantumIon server, *not* involving the end user. The key transfer message uses the `<encryption-key>` tag, with the `encoding` attribute set to the text format. Developer identification tags `<company>`, and `<uuid>` are assigned by the QuantumIon server. The enclosed value is a `CDATA`[1] array. Other tags, such as `<product-id>`, and `<version>` are assigned as needed by the developer, and are ignored by the QuantumIon server.

```
<encryption-key encoding="base64">
  <company value="Fictitious Gate Corp, Inc"/>
  <uuid value="48f784bc-deb1-4bf2-a1f7-4d12234f2862"/>
  <![CDATA[
    S/zpO96MIj0BjDKjQZq2xyY16amEK5NqkaTy7PGwgHU=
  ]]>
</encryption-key>
```

The encrypted gate data has two parts: an unencrypted header, and an encrypted gate segment. This raw XML file can be passed from the third-party developer to the end user.

[1] `CDATA` arrays are treated as character literals by XML parsers

```
<secure-header>
  ...
</secure-header>

<secure-library uuid="48f784bc-deb1-4bf2-a1f7-4d12234f2862">
    <keyHash type="sha256">
      a8add4003bf9e1eed2aca713db9bacc0d453625f001e3d5353ac748df920c994
    </keyHash>
    <cipherText cipher="aes-256-cbc">
        <![CDATA[
          U2FsdGVkX18qw/UtRX41CRVqKclulB2FrmVjBhf1MLc/rmUazjTHk34wPG2yRDUl
          MB+kcQYjGAl1GiWNtZRh5kcSHTFtFglDeBpAcBltRWYTWGwYehR2frmqKfYBYaN9
          dQdsPxFOPcnP6EwyCpXU/2sm8WYCKF4TVftNRI2Xd1uC5ZctVBfeFlPNw+hSxt/I
          DXjoC115nprvvwj/ouQUiOVgwiO1BLMuFx7fURfyJwjOHbO8dVhSLWTVFYD/kD2i
          W5/PFoIvkxhB1CxIcB8y4cYzCs7rGJJomI1DLxfYUGES9eWwOu+2jZStcPuDlqQ7
          rU2fUEkNzuqG2ZOzTOncvRA5b7RBm4gUORunOxNBFy+7BPvOkAn9nRfCCPYACr2g
          gJZecxx3WyKgiYbe4jT4P1TUvicK3aNv3h6H9Zr/Nn74uA3DIpk4BVGmOxAY4RZX
          VU1gGgsjJ/PaylLqdWw1UXZBkDk+E/NEayLY7tVauLUEIOdqxf5CtHyHrFT5So6g
          E1yfURIme+opXGHziffw0aDPDWcxrrTtfINK+fzROTKh7PIM/sm6OWoi3PgXRB/X
          tkYJe1TOu71rZBE5Xj/9ghssVTgG9dSY73O6vTWmLs0=
        ]]>
    </cipherText>
</secure-library>
```

# Appendix C

# An FPGA Primer

FPGA devices are an advanced form of programmable hardware, and provide a platform for extremely fine-grained, high-speed control of electrical hardware. The term *programmable* is somewhat misleading, since the process of defining FPGA hardware is not much like a traditional PC programming language such as C, C++, Python or Java. Such languages place an emphasis on the sequential operations used to create some function or algorithm. This sequential nature is one of the core ideas behind the theory of microprocessors: instructions are executed one-at-a-time[1].

Programmable hardware devices like FPGAs do not have a microprocessor construct (exceptions being the Xilinx Zynq family), and so the engineer is given freedom to create other digital structures. In an FPGA device, the function is defined in a manner akin to the design of digital logic hardware; in fact it is common to create dozens, even hundreds, of parallel structures, each operating independent of the others. The programming of FPGAs is performed using a so-called Hardware Description Language (HDL). One can imagine that the description of detailed digital interconnects on a sophisticated function can quickly get out of hand[2]. The two main HDL languages in use today are Verilog and VHDL. HDLs focus on two primary means of describing logic in two ways: Behavioral Logic and Combinatorial Logic. Behavioral Logic focus on the description in terms of sequential operations (similar to a microprocessor language); this is often in terms of a State Machine. State machines are often used to describe the details of a complex circuit, but are

[1] Multi-core processors, Graphics Processor Unit (GPU) processors, and multi-threaded operating systems rely on sophisticated logical constructs such as *semaphores* to break the sequential construct. See [63], Chapter 2. This background discussion's focus is on hardware parallelism and so will not consider software locking further.

[2] Even a seemingly simple function like 4-bit wristwatch processor, can contain tens of thousands of transistors

usually very small and compact, chained together to form a more complex function (such as an Ethernet controller). Combinatorial descriptions are often the means used to chain together these smaller function. An example of these two styles is shown in Section A.4.
In a static processor design, collections of transistors are interconnected to create the desired logic function. However, once the design is implemented it cannot be changed. FPGA designs exploit the symmetry of fixed logic and memory lookup tables. An example is shown Figure C.1. Consider the NAND gate, whose logic is shown in Table C.2.
The lookup table is implemented as Random Access Memory (RAM), and has two modes: read and write. In write mode the WR pin is held high, and the value of the D pin is stored at the address given by pins A0 and A1 (replacing a previous unknown state $X$). In read mode (WR pin low), the output Q is a copy of the stored value at the address given by A0 and A1. This is illustrated in Table C.1.

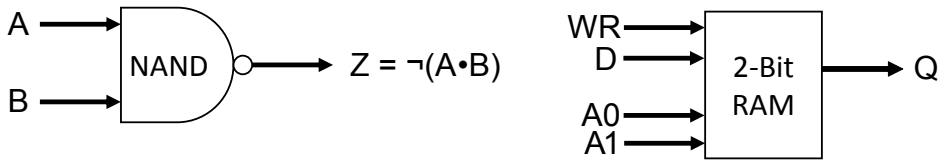


Figure C.1: NAND circuit and memory lookup table

| WR | D | A1 | A0 | RAM Contents | Q |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1XXX | X |
| 1 | 1 | 0 | 1 | 11XX | X |
| 1 | 1 | 1 | 0 | 111X | X |
| 1 | 0 | 1 | 1 | 1110 | X |
| 0 | X | 0 | 0 | 1110 | 1 |
| 0 | X | 0 | 1 | 1110 | 1 |
| 0 | X | 1 | 0 | 1110 | 1 |
| 0 | X | 1 | 1 | 1110 | 0 |

Table C.1: 2-Bit RAM Lookup Table (where 'X' indicates don't-care)

Note that the top segment of Table C.1 also shows the loading process: by providing a value of D for each permutation of A0 and A1, the memory is essentially filled (that is, completely specified). Later, the entire contents can be read out by permuting each combination of A0 and A1. When used in this way the RAM address pins A0 and A1 are inputs to a lookup table.

Now compare this lookup table with the truth table of the two-input NAND gate, as shown in Table C.2. Replacing A0 and A1 of the lookup table with inputs A and B, the same function as the NAND was created.

| A | B | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Table C.2: NAND Gate Lookup Table

This illustrates the power of the programmable hardware. The entire NAND functionality was implemented simply by the pattern loaded into RAM during the write phase. Chaining these simple lookup tables together allows any logic to be synthesized, and combining this with high-speed clocking allows sophisticated state machines and execution engines. A new function can be synthesized simply by loading a new (albeit very large) set of two-bit RAM tables, hence the term Field-Programmable Gate Array (FPGA). Practical FPGA chips use tens of thousands of such logic blocks. Unlike a CPU, where each function runs in sequence, the logic in an FPGA can be massively parallel, for example developing ten-thousand parallel addition blocks as the core for a $100 \times 100$ matrix sum, $M_{i,j} = A_{i,j} + B_{i,j}$ in a single compute cycle.

# Glossary of Terms

This document is incomplete. The external file associated with the glossary 'terms' (which should be called `output.terms-gls`) hasn't been created.
Check the contents of the file `output.terms-glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.
You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `output.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "output"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "output"
  ```

Then rerun LaTeX on this document.
This message will be removed once the problem has been fixed.